Bulletin of the Technical Committee on

Data Engineering

September 1998 Vol. 21 No. 3

IEEE Computer Society

Letters

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering (http: www. is open to all current members of the IEEE Computer Society who are interested in database systems.

The web page for the Data Engineering Bulletin is http://www.research.microsoft.com/research/db/debull. The web page for the TC on Data Engineering is http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the TC Chair

To all members of the TCDE:

There will be an open meeting of the TCDE during the 1999 Data Engineering Conference to be held March 23-26, 1999 in Sydney Australia. The time of the open meeting has not yet been decided, but it will be scheduled sometime during the conference. Professor Erich Neuhold, the vice-chairman of the executive committee of the TCDE, will chair the open meeting. All members of the TCDE are invited to this meeting.

For information about the Data Engineering Conference, see the conference web site at

`http://www.cse.unsw.edu.au/icde99/.`

Second, as a reminder, this will be the next-to-last issue of the Bulletin with paper copies automatically mailed out. The electronic form of the Bulletin will continue and we anticipate that most members will obtain their Bulletin issues from the Bulletin web site:

`http://www.research.microsoft.com/research/db/debull .`

IF YOU CANNOT ACCESS THE WEB, you must request a paper copy of the Bulletin from Ms. Tracy Woods at the IEEE CS using the email address twoods@computer.org, or the post office address

Tracy A. Woods
IEEE Computer Society
1730 Massachusetts Avenue NW
Washington D.C. 20036-1992 USA
twoods@computer.org

Betty Salzberg
Northeastern University

Letter from the Editor-in-Chief

Our New Financial Plan– Again

The TC on Data Engineering does not generate enough revenue to adequately fund the Bulletin. As TC Chair Betty Salzberg explained it in the June issue, *"we will no longer be automatically mailing out paper copies of the Bulletin to all members of the TC. This will cut down on the expenses of the TC without, we hope, causing undue inconvenience to our members, most of whom will get their copies of the Bulletin through the Bulletin web page."* This information is included in the Bulletin, as hardcopy distribution is our only sure way to reach all TC members. If you are unable to access the Bulletin via our web site and want to continue receiving it, you *must* contact Tracy Woods at the IEEE Computer Society now. The December 1998 issue is the last issue that will be sent in hardcopy to all members.

Changes to the Bulletin Web Site

The Bulletin web site has been "dressed up" a bit. Web pages are now in "vibrant" colors. But more importantly, there are two additions to the content available on the site. All issues are available in PDF format as well as in postscript. Further, begining with the March 1998 issue, individual articles of the issue are available as separate postscript files. I invite you to explore the redesigned web site and to send me your comments and suggestions.

This Issue

Database interoperability is an old subject. And while this area has not attracted much research attention over the past several years, interoperability is an area that continues to be very important in the real world of heterogenous database systems and diverse information repositories. Interoperability is frequently the way that we deal with legacy systems. So this subject will retain its importance for as far as any of us can forsee. Donald Kossman has put together an issue that includes articles from both researchers and commercial vendors. Thus, the issue provides a broad view of where we are today, and where we might be going. I thank Donald for his hard work in bringing the issue to fruition.

David Lomet
Microsoft Research

Letter from the Special Issue Editor

The ability to interoperate is becoming an essential feature of modern data processing systems. First of all, interoperability is important because usually not one software module meets all the data processing requirements of a whole company. As a result, companies buy and install several components from potentially different vendors (including different database vendors), and these components need to interact in order to serve the company's purposes. One particular situation is that a company buys a new system for, say, sales and distribution and this system must coexist and interact with an old system for, say, stock management. Interoperability is also important to support e-commerce or automatic supply chain management in which different data processing systems of different companies must interact (e.g., on the Internet).

Of course, it will never be possible to plug any two systems together and make them cooperate seamlessly, but there has been significant progress in the last couple of years due to developments such as Java, CORBA, OLE-DB, and object-oriented and object-relational database technology. The purpose of this issue is to give an overview of the state of the art. The bulk of this issue consists of three papers that describe the interoperability concepts and products of three major database vendors: IBM, Sybase, and Informix. While the products of these three vendors have a great deal in common (e.g., use of the abovementioned standards and technology), there are significant differences in the approaches taken by these three vendors.

The first paper describes the IBM approach. The paper shows that IBM provides a whole range of products and features in order to integrate data from external data sources. Specifically, IBM's DB2 database products support (1) *table functions* which can be used to access external data just like (standard SQL) tables; (2) *extenders* which can be used to integrate non-standard data (e.g., geospatial data); (3) *DataLinks* which can be used to access data stored in ordinary files; and (4) *wrappers* which can be used to integrate data from complex external data sources.

The second paper gives an overview of the interoperability concepts implemented as part of the Sybase Adaptive Server products. Compared to DB2, Sybase does not attempt to integrate all kinds of (non-standard) data types and specializes instead on the integration of the most important data types: text, time series, and geospatial data. Furthermore, Sybase implements a special series of query optimization techniques in order to deal with external data and these techniques are also described in the paper. Finally, Java plays a dominant role in Sybase's strategy, and the paper, therefore, also describes SQLJ, a new standardization initiative to embed SQL into Java.

The third paper presents Informix' virtual table interface (VTI). The VTI complements Informix' data blades and object-relational features and plays a similar role than IBM's table functions; i.e., virtual tables make (non-standard) external data appear as (standard) relational tables. The paper describes the details of the interface including support for query optimization and update operations.

The fourth paper describes the design of a *data broker*. A data broker is a tool that can be used to build interoperable systems; a data broker could, for example, be used to connect IBM, Sybase, and/or Informix systems with other database and application systems. Putting it differently, a data broker provides the glue that holds the individual components together. One particular aspect is to achieve good performance, and the paper describes performance issues and experiences with scalable data brokers that perform well in the presence of hundreds of concurrent users and various different component databases.

The fifth paper describes the `Concert` research project at ETH Zurich. The approach devised in that project differs significantly from the general approach taken by the commercial products. The idea is to *export* the functionality of a database system. That is, rather than *importing* the external data and processing the data in the database system, the `Concert` project shows ways to move the database functionality to the data so that the data can be processed in-place. Specifically, the paper addresses issues that arise in the physical design and for transaction management.

Donald Kossmann
University of Passau

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.

The Sybase Architecture for Extensible Data Management

Steve Olson Richard Pledereder Phil Shaw David Yach
{*olson, pleder, phil.shaw, yach*}*@sybase.com*

1 Overview

Enterprise integration—unification of all of an organization's information resources, from the mainframe to the desktop, into a seamless, coordinated, easily accessed corporate asset that appears and works as one virtual system—has been an elusive goal for many large organizations. To help achieve this goal, and to help customers solve business problems and achieve competitive advantage, Sybase has introduced a comprehensive strategy for enterprise computing called the Adaptive Component Architecture (ACA). This architecture is based on the requirements of today's business computing environment, which can be summarized as follows:

- Use of industry-standard components
- Rapid application development
- Delivery of data in the right form, to the right place, at the right time
- Minimized complexity for both end users and developers

1.1 Adaptive Component Architecture

Based on open component logic, comprehensive development tools, and optimized data stores, Sybase's Adaptive Component Architecture[7] (ACA) provides a multi-tiered framework designed to manage and deploy components across the distributed computing environment. ACA features an Application Server tier and a Database tier. The Application Server tier supports component-centric computing (Jaguar Component Transaction Server) and page-centric computing (PowerDynamo). Sybase Adaptive Server forms the strategic Database tier.

The Adaptive Server brings existing Sybase DBMS products – SQL Server, Sybase IQ, and SQL Anywhere – into this unified architecture. Adaptive Server features optimized data stores for delivering predictable high-performance management of traditional and complex data within many different types of applications. It also offers a single programming and query interface across all the data stores, using Transact-SQL and standard components, including JavaBeans, running in the server. It also supports specialty data stores, which share a unified programming and operational model. Incorporated into the Adaptive Server is a component integration layer, which enables distributed access to each of these data stores, including multi database distributed transactions.

1.2 Evolution of Sybase Data Access Middleware

In the early 1990s, Sybase management and engineering came to the conclusion that the Sybase RDBMS was not going to displace existing DBMS systems in customer accounts. Consequently, a strategic decision was made to adopt a policy of *coexistence*, rather than *displacement*. This decision led to a series of interoperability products, beginning with the Sybase Open Server. The Open Server allowed Sybase customers to write their own gateways that would inter-operate with the Sybase RDBMS and with Sybase Open Client application programming interfaces (API's). This toolkit was also the basis for the first Sybase gateway, called the Net Gateway, which enabled easy access to CICS transactions on MVS and IMS, and to DB2 using dynamic SQL.

Later, a series of additional gateways were introduced, all based on Sybase Open Server, which provided application and database interoperability with a variety of non-Sybase RDBMS, including Oracle, Ingres, Informix, Rdb and RMS files on VMS systems

While these gateways solved the primary problem of obtaining access to non-Sybase database systems using Sybase APIs, not all databases are created equal. An application needing access to Oracle could use Sybase APIs to access an Oracle gateway, but needed to use Oracle's PL/SQL syntax in order to do so. Additionally, if an application needed access to two or more database systems, it needed to use the specific SQL syntax associated with each, and it needed to manage SQL joins between two or more systems.

The OmniSQL Server product was conceived and resulted in Sybase's initial Multi-DBMS (MDBMS)[6]. This product enabled applications to interface with a single server even though access to many separate database systems was necessary. The database specific access mechanisms were hidden from the application by the OmniSQL Server. To an application, OmniSQL Server appeared to have the same look and feel as the Sybase SQL Server. The resulting *location transparency* was a key feature of the first release of OmniSQL Server. Additionally, the semantics of Sybase Transact-SQL were enforced. This *functional compensation* ensured that the behavior of the application could remain consistent, regardless of the nature of the data source(s) involved in various queries and transactions initiated by the application.

1.3 Sybase Adaptive Server

Sybase Adaptive Server (formerly known as Sybase SQL Server) ships in two variants: Adaptive Server Enterprise and Adaptive Server Anywhere. Both variants contain the Component Integration Services[8] which encapsulate the support for distributed, heterogeneous data management. Figure 1a illustrates the relationship between the Sybase Adaptive Server Enterprise, Component Integration Services, and external Sybase and non-Sybase database systems. Figure 1b illustrates the relationship between Sybase Adaptive Server Anywhere, Component Integration Services, and external Sybase and non-Sybase database Systems: Note the use of standard JDBC and ODBC APIs in case of Adaptive Server Anywhere.

The following sections will take a closer look how Adaptive Server supports distributed query processing, specialty data type support, and Java extensions.

2 Distributed Query Processing using Component Integration Services

Performance is the leading source of concern expressed by most users of distributed systems. Component Integration Services (CIS) addresses many of these concerns by focusing on two separate aspects of distributed query processing:

- Query decomposition – analyzing query syntax and determining the amount of work to be pushed to remote sites for processing
- Query optimization – analyzing query syntax to establish optimal join strategy and join order

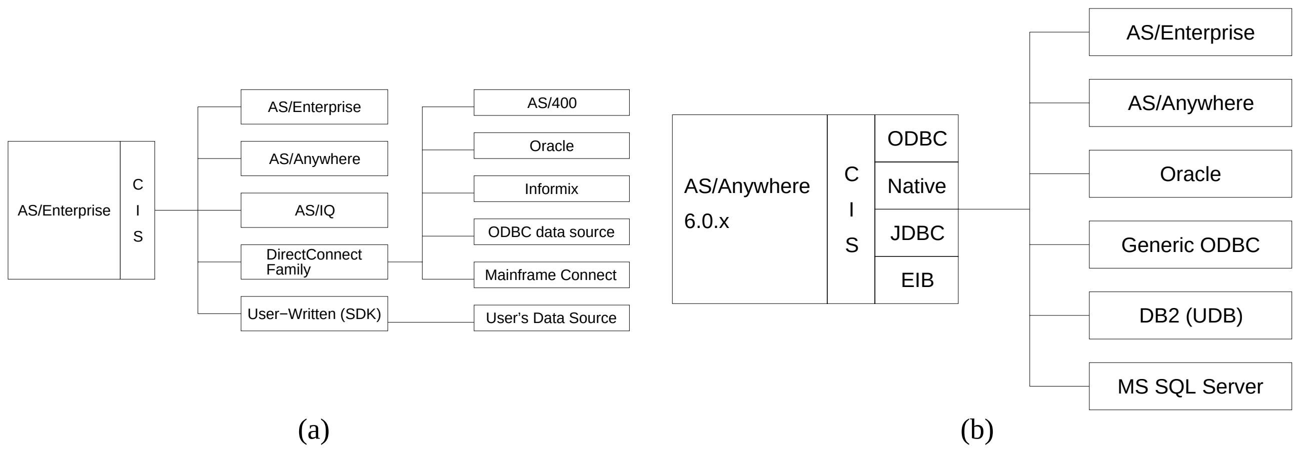

Figure 1: ASE, CIS and Distributed Access

2.1 Query Decomposition

CIS evaluates query trees at two stages of query processing:

- Pre-optimization – before the query optimizer is invoked to analyze various permutations represented within the query
- Post-optimization – after optimization, but before plan generation and execution, to determine if more of the work represented by the statement can be forwarded to a remote location for execution.

2.1.1 Pre-Optimization Query Decomposition

CIS will intercept query processing after the preprocessing stage has completed, but before query optimization is invoked. Preprocessing is necessary in order to perform view resolution. A decision will be made at this stage regarding the level of functional compensation that will be required. There are two questions that must be answered:

- Is every table represented within the SQL statement located on the same remote server?
- If so, is that remote server capable of handling all of the syntax/semantics represented by the statement?

If the answer to both of these questions is yes, then a query plan will be produced which will allow CIS access methods to reconstruct the entire statement and pass it to the remote server. Results will then be routed according to the needs of the statement (back to the client, inserted into another table, assigned to a local variable). This capability is referred to as *quickpass* mode. If the answer to either if these questions is no, then the query trees are passed to the query optimization phase.

2.1.2 Post-Optimization Query Decomposition

In many cases, *quickpass* mode cannot be selected because more than one remote server is involved in the query, or the needs of functional compensation dictate that some of the work must be performed locally, so as to preserve the expected semantics of the query.

Nevertheless, CIS will attempt to re-analyze portions of the query tree to determine if branches can be used to generate sub-select statements that can be forwarded to a remote server. This is particularly useful in the case of **union** operators - one side of a union may contain a query referencing DB2 tables, for example, while another side may contain a query referencing Oracle tables. In this case, the entire select on each side of the union can be reconstructed and sent to the affected database, and the results merged locally by CIS, according to the needs of the union operator (**union** or **union all**).

2.2 Query Optimization

The primary roles of the query optimizer when distributed joins are implicit in the query syntax are to determine access cost for each remote table and to determine join strategy.

2.2.1 Remote Table Distribution Statistics

Without proper cost information for remote table access, it is not possible to select a correct join strategy. Consequently, a means of obtaining distribution statistics for remote tables is needed. This is done in the Adaptive Server (and OmniConnect) through the **update statistics** command. If the object of the command is a local object representing a remote table, the data associated with remote indexes is requested, and processed locally as if the data was derived from local index pages. The resulting distribution histogram[9] and row count information is stored in local system catalogs for later use by the query optimizer.

2.2.2 Query Optimizer Inefficiencies

The primary problem facing the distributed query optimizer is that of calculating accurately the cost of accessing objects across a network. This is an especially important problem to solve when a query involves a combination of local and remote tables, or when the query involves tables residing on two or more separate database systems. If the costing is wrong, then it is possible that the tables that require network access will be positioned improperly in the resulting query plan, resulting in unacceptably high amounts of cpu and elapsed time to resolve the query. For example, in this query of local table L and remote table R:

```
SELECT * FROM L, R WHERE L.i = R.i
```

Local table L could end up being the outer table and remote table R could end up being the inner table of a nested loop join strategy, accessed with a remote query, as follows:

```
for each row of L:
SELECT ⟨ column_list ⟩ FROM R WHERE i = ?
```

As the number of rows in table L increases, the worse the results become. The cost of remote access must be taken into account, so that in this example, the remote table could be positioned as the outer-most table of the nested-loop join strategy, thereby eliminating the network cost for each member of the inner table.

Secondly, during post-optimization processing, peephole optimization can combine the queries to remote tables on the same server into a single sub-join. For this to occur, those remote tables must be adjacent in the plan and be directly connected. For all this to line up properly requires sufficient intelligence in the optimizer to recognize the location of remote objects, and to recognize that join clauses are possible to be pushed to remote locations.

These issues have been addressed in the Adaptive Server by the following changes to the query optimizer:

- A remote access cost formula to more accurately estimate remote query overhead has been implemented.
- Transitive closure has been introduced for joins to ensure remote tables located on the same server are connected at the first order when possible.
- The cost of access overhead for adjacent (in the query plan) remote tables located on the same server has been re-evaluated to reflect that the join between the tables will be processed remotely.

2.2.3 Evaluating the Cost of Network Overhead

Efficient execution of distributed queries requires that the remote tables be accessed once during the processing of a query (or even better, accessed as a group, once, via a remote sub-join). The largest cost is network access, not disk access. The existing cost formula takes into account estimates of logical and physical disk reads, and physical writes in the case of inserts. Access to remote tables also requires an estimate of the network i/o.

The following cost formula has been introduced into the query optimizer to calculate the cost of remote access:

```
    (3 + floor((rows-1) / 50)) * WEXCHANGES
```

An exchange is one elemental network interaction: send a message to a remote server, get a reply. For a remote **select**, there is one exchange to open the cursor, one to return up to 50 rows, and another to close the cursor. If more than 50 rows are returned, another exchange is required for each additional set of 50 rows. We set the constant WEXCHANGES at 100 (nominally milliseconds) relative to WLOGICAL at 2, WPHYSICAL at 18. The estimated cost for disk access (logical i/o's, physical i/o's), although calculated as if the remote tables are local, is retained as an estimate of the cost of this work on the remote server.

With this formula in place, the estimated cost of accessing a remote table repeatedly via a parameterized **select** statement skyrockets, as it should. The cost of scanning it repeatedly is astronomical! But, the cost of placing that table as the outermost table in the plan and accessing it once by efficiently streaming its results over the network becomes very attractive.

2.2.4 Peephole Optimization and Remote Sub-joins

In many cases, reformatting is avoided by peephole optimization performed on the query plan. The query plan is evaluated to locate connected remote tables that reside on the same remote location and are adjacent in the plan. These tables are gathered into a single remote query, which performs a sub-join remotely. Only the result of the remote sub-join is retrieved over the network.

In order for this peephole optimization to be activated, the remote tables need to be adjacent in the plan and be connected at the first order. More modifications to the optimizer were required to make this fall out reliably.

First, transitive closure for joins was added to query processing. This algorithm finds all specified relationships between the tables by walking the tree in search of join clauses. Then, all implied but unspecified relationships are derived and added to the query tree as (redundant) join clauses. Now all relationships between tables are directly specified. For example:

```
    SELECT * FROM R1, R2, R3
    WHERE R1.i = R2.i AND R1.i = R3.i
```

This implies that R2.i = R3.i. Transitive closure for joins adds this join clause to the tree giving the query:

```
    SELECT * FROM R1, R2, R3
    WHERE R1.i = R2.i AND R1.i = R3.i AND R2.i = R3.i
```

Remote query costs are much lower when tables residing on the same server can be accessed in a remote sub-join. During join costing, adjustments are made to the cost of remote access when costing a remote table that is ordered immediately before another remote table on the same remote server. If these tables are connected and both tables are being evaluated with the nested iteration access method, then the cost estimate is modified as follows.

For each table pair in the current work plan permutation that represent remote tables at the same location, and are directly connected:

- Change the remote access cost of the inner table to 0.

- Calculate the remote access cost of the outer table as the result of the sub-join of the two tables.

In other words, estimate the number of rows returned from the remote sub-join of the two tables and cost that instead of costing individual remote sub-queries. This algorithm cascades over successive remote tables in the work plan permutation.

The net effect is to favor a query execution plan that groups remote tables that are located on the same remote server so they can be combined into a single remote sub–join. Also, regardless of whether there is a single remote table or a group, remote tables are highly favored as the outer table of the join, reformatted to a work table, or selected for a merge-join and, hence, accessed only once during the query.

3 Specialty Data Stores

The high-technology press and analyst community has generated considerable speculation regarding support for extensibility in relational database management systems and real-world requirements for user-defined data types. Discussions with customers reveal that they actually need a solution for a small, well-defined set of problems. This section provides a background on Specialty Data Stores and Sybase's strategy for supporting them in the Adaptive Server.

3.1 Requirement for Special Data Types

Increasing demand for access to more information available within an enterprise is driving requirements for the RDBMS to support other data types in addition to alpha-numeric. This demand can be met with a small number of specific data types that can dramatically improve the utility of client/server applications for the end user, and which can simplify the development of such applications.

The emergence of the Internet has also increased the focus on these specialty data stores. Although static in content, the information on most Web sites today includes a rich set of text and image data. The evolution of client/server applications to the Internet enhances the need for RDBMS to support these rich data types.

The specific data types that have gained mind-share for their potential are:

- Text – in the sense of ”full-text” search and retrieval
- Time Series – essentially a compact array with intrinsic awareness of time concepts
- Geospatial – spatial semantics specific to geographical data

There are also specializations of these types, such as a fingerprint comparison application of the image content-type, but these tend to be vertical-market specific. Furthermore, there is a potential for application vendors to build their own types to facilitate their application design and development.

A primary support issue for these data types is that they must be integrated into the RDBMS. From the application and systems management perspectives, they should seem as if they are native features. Indeed, in some cases, the RDBMS can be extended to provide these capabilities natively. In other cases, in terms of performance and integrity, it will be most effective to support existing best-of-breed technologies and provide integration points into the server. A single request of the server should be able to access both native and federated services.

3.2 Text

Requirements for text come from the need to locate documents that pertain to transactions. Users need to search documents in their native formats (ASCII, HTML, XML, Word, WordPerfect, etc.) using a variety of powerful techniques, including ”topical” searches. For instance, a trader might want to see documents relating to microprocessor developments in a search for ”personal computers.”

Applications that search text need to find documents wherever they are located. The RDBMS should be able to search documents stored in the database. However, it is often necessary or more efficient to leave documents outside the database and index them along with documents stored inside the DBMS.

The Sybase solution for full-text search requirements involves integrating software from Verity, Inc., into a separate search engine which is accessed by Component Integration Services when text search operations are requested by the client application. The search engine has knowledge of the Adaptive Server data store, and uses its data to construct text search metadata. This metadata (data about data) is used to allow the text search engine to rapidly scan through text indices to resolve a full-text search query.

3.3 Time Series

Several vertical market segments process data that has an intrinsic time attribute. Stock prices are associated with a ticker value, a day, an opening of the market, a closing of the market, etc. Each of these events gives different meaning to the price. A series of prices occur at regular intervals, such as a series of daily closing prices.

Simple storage and retrieval of such a series might be done in a standard table with two columns, but the time column is actually redundant. All that is really necessary to associate a given price with its relevant date is the starting date of the series and the offset into the list of prices. A time series data type is structured according to this notion of time.

Furthermore, operations on the series require the semantics of frequency conversions. For example, to produce a series of weekly closing prices, a conversion must be done on the series to locate the Friday values. To produce a five-day moving average, a function must process the series in rolling groups of five market days to get the resultant series.

Storage and indexing efficiencies must be considered for time series. Many series are sparsely populated, and most dedicated time series engines do not store the missing values. Nevertheless, the series can be quickly indexed via the location of values in the series – a concept foreign to relational table storage mechanisms.

The Sybase solution for handling time-series data involves a partnership with Fame, Inc., which has provided a Specialty Data Store that can be accessed by Component Integration Services in order to provide support for this data type to client applications.

3.4 Geospatial

Government bodies responsible for managing our public physical assets such as roads, utility infrastructures, emergency services, and natural resources have used mapping technologies from Geographic Information Systems (GIS). These tools process data accessed by latitude and longitude, and sometimes elevation. This data is typically called geospatial data.

Geospatial data has traditionally been processed in the application or client software. The data servers and DBMSs have had no understanding of the semantics, or indexing methods for geospatial data. However, there is a clear trend toward bringing the benefits of client/server and multi-tier computing to GIS. Providing data types, predicates, and indexes specific to geospatial data is an emerging DBMS requirement.

The Sybase solution for handling geospatial data types involves a partnership with Vision International, Inc., which provides a Spatial Query Server. Spatial Query Server (SQS) is an Open Server application that allows an application to include geographic constructs in SQL statements. SQS adds three enhancements to the Adaptive Server: spatial data types (e.g., point, line, polygon), spatial operators (e.g., inside, intersect), and two-dimensional spatial indexing. The SQS decomposed each query it receives, changes the enhanced SQL language, called Spatial SQL, into Sybase Transact-SQL, then sends the query to the Adaptive Server. The SQS then post-processes the result set coming back from the Adaptive Server and relays the final results back to the client.

4 Extensibility with Java

Java is quickly being adopted as a *3rd generation object-oriented language*. Furthermore, Java is beginning to play a major role in providing extensibility to Database Servers and Application Servers. The reasons for that are numerous:

- *Ubiquity:* Java is inherently portable.
- *Safe Component Integration:* Java supports strong typing, separation of implementation from interface and, most importantly, automated memory management which obviates the need for pointer manipulation.
- *Ease-of-Deployment:* Java treats code and data uniformly.
- *Reusable Components:* Java can run on any tier and Java objects can easily be exchanged between tiers.

4.1 SQLJ

Enterprise Computing is inherently multi-vendor. For this reason a set of *uniform APIs* is required to ensure interoperability among multi-vendor solutions. These APIs must be available across *all tiers* and support:

- Client Access,
- Scalable Component Servers,
- Warehousing Engines and
- OLTP Database Servers.

The SQLJ standard developed by the SQLJ consortium describes ways for Java to be used with SQL. The SQLJ effort is driven by major industry vendors such as Oracle, Sybase, IBM, JavaSoft, Tandem, Informix and others. SQLJ specifies the syntax and semantics for Java with Embedded SQL, Java Stored Procedures, Java UDFs and Java Data Types. SQLJ is well integrated with the JDBC API. The SQLJ approach to routines and data types is sometimes referred to as *Java Relational*[3]. For a complete description of the SQLJ standard, we defer to the reference material [1]. Sybase has been one of the major contributors to the SQLJ consortium.

4.1.1 JDBC

JDBC[2] provides Java programmers with a Call Level Interface to perform dynamic SQL operations. JDBC 1.0 is now widely supported by database systems and tools. Version 2.0 of JDBC has added features that allow the application and the database to exchange structured data, including Java objects.

4.1.2 Embedded SQL

SQLJ Embedded SQL allows Java programmers to include SQL clauses in their Java programs. The SQLJ Translator is then used to perform design time validation of the SQL statements and replace them with Java code. SQLJ Embedded SQL also specifies a vendor neutral runtime infrastructure and a vendor neutral representation of the preprocessed SQL statements.

4.1.3 Stored Procedures and UDFs

SQLJ Stored Procedures and SQLJ User-Defined Functions allow the programmer to write these routines in Java and install them through JAR files into the database. SQL operations are expressed by either coding Embedded SQL or raw JDBC calls. Client applications use JDBC, ODBC, etc. to invoke these procedures. By contrast, Java UDFs may be invoked from within SQL DML statements.

4.1.4 Data Types

With SQLJ Data Types programmers may use Java Classes to declare the types of columns, procedure parameters, etc. Java represents a very attractive choice for expressing complex types in a database, due to features like object-pass-by-value or the sandbox execution model. SQL DML statements are extended to permit the access to Java methods for relational operations such as project, filter, etc.

4.2 Java and Sybase Adaptive Server

One of the key features of the recently released Sybase database product Adaptive Server Anywhere[4] is the support for SQLJ-compatible Java Stored Procedures, Java Functions and Java Data Types.

4.2.1 Implementation Aspects

In order to ensure efficient integration of database processing with Java computing, Sybase Adaptive Server hosts a Java Virtual Machine (JVM). The JVM is responsible for performing all Java operations such as execution of *static methods* or invocation of *instance methods*.

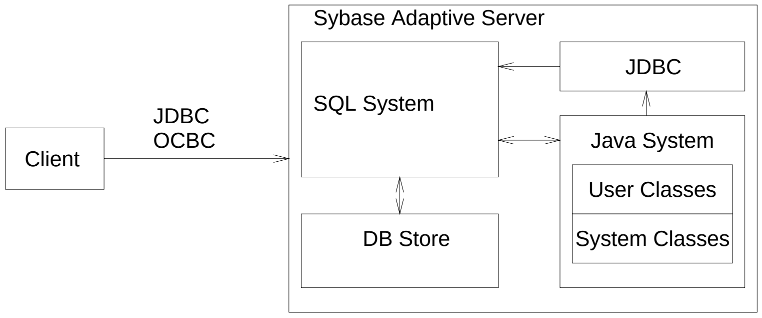

Figure 2: Integrating a Java System with SQL

When a SQL operation involves Java, the SQL System of Adaptive Server makes use of the Java System for operations such as:

- Instantiation of Java objects, either from persistent storage or from client transfer,
- Invocation of instance methods on Java objects,
- Invocation of static methods on Java classes.

Compiled Java code (classes) can be executed within the server environment using SQL statements. The use of a Java Virtual Machine in the database environment ensures that Java in the database fully conforms to Java standards and expectations. Public class and instance methods can be executed. Classes may inherit from other classes. Packages and the Java API are supported. Finally, access to protected, public and private fields is controlled. Every Java class installed in a database becomes available as a data type that can be used as the data type of a column in a table or a variable. An instance of a Java class (a Java object) can be saved as a value in a table. Java objects can be inserted into a table, SQL select statements can be executed against the fields and methods of objects stored in a table, and Java objects can be retrieved from a table. The use of Java in no way adversely or unnaturally alters the use of SQL statements or the way other database schema elements operate. An internal, high-performance JDBC driver executing native functions lets Java objects access SQL data within the database environment.

We expect users to deploy complex Java processing logic into Adaptive Server. An interactive, source-level debugger is provided to assist users with isolating problems in the deployed Java code. Furthermore, high performance access matters to database users. For this reason, attribute-level indexes may be created on columns that are declared as a Java type.

4.2.2 Java Stored Procedures

Sybase Adaptive Server was the first commercial database server to support Stored Procedures based on Sybase Transact-SQL[5]. Transact-SQL (TSQL) enables enterprise customers to host business logic close to the business data, thus improving performance and enabling Client/Server computing.

While Stored Procedures are now a common place feature for most commercial database systems, they have forced business programmers to give up on portability and modern language features in favor of performance. To address this, Sybase approached the SQLJ consortium with an initial proposal for a standard way to use Java as a Stored Procedure language.

Adaptive Server Anywhere Version 6 provides a first implementation of SQLJ-style Stored Procedures. This enables true portability of Stored Procedures across different DBMSs and maintains the capabilities of regular SQL Stored Procedures. SQLJ Stored Procedures are callable from ODBC, from JDBC, from other SQL Stored Procedures and directly from Java.

Java Static Methods Based on the SQLJ standard, any Java *static method* is callable as a stored procedure. Initially only parameter types and result types that are mappable to SQL types will be supported. However, the architecture is extensible to support arbitrary Java types. The body of the static method may use JDBC or SQLJ Embedded SQL to perform SQL statements.

The following Java class **Regions** implements two static methods, **region** and **correctStates**. The **region** method maps a state code to a region number. The **correctStates** method performs an SQL update to correct the state codes.

```
public class Regions {
    // region will be called as a function
    public static int region(String s) throws SQLException {
        if (s == ”VT” || s == ”NH” ) return 1;
        else if (s == ”GA” || s == ”AL” ) return 2;
        else return 3;
    }

    // correctStates method will be called as a stored procedure
    public static void correctStates (String oldSpelling,
                                      String newSpelling) throws SQLException {
        Connection c = DriverManager.getConnection(”JDBC:DEFAULT:CONNECTION”);
        PreparedStatement stmt = c.prepareStatement (”UPDATE emps SET state = ? WHERE state = ?”);
        stmt.setString(1, newSpelling); stmt.setString(2, oldSpelling);
        stmt.executeUpdate();
        return;
    }
}
```

Java Stored Procedures may use JDBC or SQLJ Embedded SQL to perform database operations. In this example we are using JDBC, so we require access to a JDBC connection object before we are able to perform any DML operations. In the SQLJ standard, this is accomplished by establishing a connection to a special database URL: *“JDBC:DEFAULT:CONNECTION.”* This URL effectively returns a connection to the implicit session context that the Stored Procedure executes in.

The **correctStates** method may then be invoked as a Stored Procedure from JDBC or ODBC. The **region** method may be invoked as a User Defined Function as illustrated below

};

```
select name, region(state) as region
from employees
where region(state) = 3
```

Any *Java static method* whose parameter types and resultset types are mappable to JDBC types may be called as a Stored Procedure. The architecture, however, is extensible to support arbitrary Java types. The body of the static method may use JDBC or SQLJ Embedded SQL to perform SQL statements.

Installing Java Stored Procedures Java Stored Procedures are installed into Adaptive Server Anywhere by installing a Java Class file or a JAR file. ASA automatically examines each Java Class for *static methods*. These methods are then made available as Java Stored Procedures. The Java reflection mechanism is used to determine the names, methods and signatures of the Stored Procedures.

ResultSet Processing Ordinary Stored Procedures may return result sets that are neither parameters nor function results. SQLJ Stored Procedures model ResultSets as follows. An additional clause in an installation descriptor associated with the Class file specifies that the procedure has result sets. Such a SQLJ procedure may be defined on a Java method with a result set as return value.

4.2.3 Data Types

Adaptive Server Anywhere supports the use of Java as SQL data types for defining columns of SQL tables and views. The advantage to the SQL programmer is that Java provides a simple, robust and ubiquitous type extension mechanism. Java features such as inheritance and encapsulation are immediately available. The advantage to the Java programmer is that the native support of Java objects in a relational SQL database obviates the need for mapping Java objects to scalar types or BLOB data types.

Here is an example of a Java class that we will use as SQL data type:

```
public class Address implements java.io.Serializable {
        public String street;
        public String zip;
        // A constructor with parameters
        public Address (String S, String Z) {
                street = S; zip = Z;
        }
        // Return a string representation of the full address
        public String toString( ) {
                return "Street= " + street + " ZIP= " + zip;
        }
};
```

Here is an example of a subclass derived from the **Address** class:

```
public class Address_2_Line extends Address implements java.io.Serializable {
        public String line2;
        // A constructor with parameters
        public Address_2_Line (String S, String L2, String Z) {
                street = S; line2 = L2; zip = Z;
        }
        // Return a string representation of the full address
        public String toString( ) {
                return "Street= " + street + " Line2= " + line2 + " ZIP= " + zip;
        }
};
```

Column Definitions Once a Java Class has been installed into Adaptive Server, it may be used to declare the type of table columns. The following example creates an **employee** table which contains the **home** column declared as type **Address** and the **mailing** column declared as type **Address_2_line**.

```
createtable employees (
        name          varchar(30),
        home_addr   Address),
        mailing_addrAddress_2_Line);
```

Insert Operations Regular SQL data manipulation operations are used to operate on Java Table Columns. For example, here is how a row is inserted into the **employees** table:

```
insert into employees values (
        'Bob Smith',
        new Address('432 Elm Street', '99782'),
        new Address_2_Line('PO Box 99', 'attn: Bob Smith', '99678'));
```

When the SQL system encounters the **new** keyword, it will internally instruct the Java VM to invoke the constructor method of the target Java class. In the example, it will invoke the constructor method of the **Address** class. This creates a new instance of type **Address**. The SQL system then effectively writes the value of this newly created instance to persistent store.

Retrieval Operations The user may retrieve an entire Java object or may project individual fields. The following example shows a SELECT statement where individual fields of Java objects are returned. Adaptive Server supports regular Java syntax for accessing instance fields and instance methods within a SQL statement.

```
select name, home_addr.zip, home_addr.street, mailing_addr.toString()
from employees
where home_addr.zip <> mailing_addr.zip;
```

When the SQL system in Adaptive Server encounters a reference to a column of type Java, it will read the value of the object instance into the Java VM and maintain an object reference to the instance. Subsequently, when the SQL system needs to access a field or invoke a method on this instance, it will dispatch to the Java system by passing that object reference.

Update Operations Update operations may be performed on individual fields or on the entire Java object. The following shows how the user might modify a field of a Java object.

```
update employees
        set home_addr>>zip = '99783'
        where name = 'Bob Smith'
```

4.2.4 Outlook

Our experience has been that database extensibility based on Java technology is readily accepted by database users. In addition, Java provides a built-in component model, JavaBeans, which facilitates the deployment of reusable components into the database server. Therefore, the natural progression will be for Adaptive Server to natively host scalable and transactional components based on the Enterprise JavaBeans model. Finally, we expect 3rd party vendors to provide a series of added-value components that are written in Java and that can now be safely installed into Adaptive Server.

5 Conclusion

The Sybase Adaptive Server family provides the ability to integrate enterprise information systems through the database extensibility mechanisms of Component Integration Services and Java. CIS provides intelligent distributed query processing capabilities for remote data access, and also provides the enabling technology for Specialty Data Stores. Both of these extend the reach of the Adaptive Server to enterprise-wide, legacy systems as well as to object and non-relational processing systems.

The creative use of Java in the database has provided a means for Sybase to implement support for abstract data types and user-defined functions, as well as to provide an object-relational capability within a database management system that has historically been relational only.

These extensibility mechanisms enable the Sybase Adaptive Server to co-exist in a heterogeneous world, and simplify the task of application development by presenting a uniform and consistent view of enterprise-wide systems. In addition, the extensibility features of the Adaptive Server open the door to whole new classes of applications that can take advantage of the extensibility for use in vertical markets.

Acknowledgements The authors would like to recognize everyone who contributed to the Adaptive Server Enterprise and Adaptive Server Anywhere products. The authors also recognize the efforts of the SQLJ consortium.

References

[1] SQLJ Specifications. Contact `Phil.Shaw@sybase.com.`

[2] JDBC Specification. `http://java.sun.com/products/jdbc.`

[3] Java and Relational Databases: SQLJ. SIGMOD Tutorial, G. Clossman, J. Klein, P. Shaw, R. Pledereder, M. Hapner, B. Becker, ACM SIGMOD 1998 Proceedings.

[4] Sybase Adaptive Server Anywhere V6.0, User Manuals.

[5] Sybase Adaptiver Server Enterprise Transact-SQL User's Guide, User Manual, Document ID 32300-01-1150.

[6] DB Integrator: Open Middleware for Data Access, R. Pledereder, V. Krishnamurthy, M. Gagnon, M. Vadodaria, Digital Technical Journal, 7(1), 1995.

[7] Technology and the Search for Competitive Advantage. `"http://www.sybase.com/aca/whitepaper.html."`

[8] Sybase Component Integration Services User's Guide for Adaptive Server Enterprise and OmniConnect, User Manual, Document ID 32702-01-1150.

[9] Histogram-Based Solutions to Diverse Database Estimation Problems. Y. Ioannidis and V. Poosala, Bulletin of the Technical Committee on Data Engineering, September, 1995, IEEE Computer Society.

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 tear-down 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.

Solving the Data Inter-operability Problem Using a Universal Data Access Broker

Mike Higgs Bruce Cottman
I-Kinetics, Inc.
{*mhiggs, bruce.cottman*}*@i-kinetics.com*

1 What is Data Inter-operability?

Stored data has been fundamental to human enterprise ever since the invention of clay tablets, or perhaps even as early as primitive cave paintings. All stored data requires the definition of persistent representations for that data, and corresponding interfaces to access and use that data. Once invented, each representation of persistent data must inevitably inter-operate with humans or mechanisms that use other persistent representations. None of this is new to the human experience, yet the Biblical Tower of Babel is as nothing compared to the bewildering data interoperability problem brought forth by the computer in just fifty years. It's perhaps hard to imagine Alexander halting his armies whilst critical data was translated by scribes from obsolete forms. But no great leap at all to imagine a modern day corporate Alexander stopped in his tracks because his CIO simply cannot get one enterprise system to inter-operate in the necessary manner with data produced by another enterprise system. Or perhaps because his CIO cannot achieve the required data inter-operability at the transaction speeds required by some new business campaign.

2 Historical Approaches to Data Inter-operability

Like much else we invent, a human tendency is that the solutions we form to the problems of data inter-operability have human parallels. Thus many of the most successful solutions parallel translation or interpretation of human languages. To achieve data inter-operability stored data is transformed from one representation to another. However like translation of human languages this solution falls short of the ultimate. We want the translator to be dynamically available. We want the translator not just to literally transpose but to deduce or infer semantics. We want the translator to understand all representations, including new variants or dialects as they are added. We want the translator to operate in real time (*on the fly*). Last but of course not least, we want to inter-operate with the translator in the representation of our choice, not one that the translator imposes on us because it will only support certain combinations of representations.

Another human tendency is to continually attempt to solve a problem by inventing the ultimate representation for persistent data. For the problem of data inter-operability this tendency manifests as a series of generations of database technology with corresponding persistent representations for stored data. In theory translating all required data into a new persistent representation will indeed solve the problem of data-interoperability. But unfortunately this flies in the face of both historical and economic reality. The bulk of existing stored data will never be translated into the new persistent representation even if that persistent representation were viable in all problem domains, which it will not be. The truth is that each new persistent representation adds to the Tower of Babel, and the very success of a persistent representation in specific problem domains ensures that it enlarges the problem it is intended to solve rather than reduces it.

A more practical approach, which has been attempted from time to time, is not to invent the ultimate persistent representation into which all stored data must be transformed, but rather the ultimate interface by which all stored data must be manipulated. A recent incarnation of this approach is Microsoft's OLE-DB Universal Data Access [OLE97]. Of course this approach too is flawed, because all interfaces are to some extent problem domain specific, or else so pithy as to be useless. Yet this approach is also more useful, because from time to time different interfaces may become de-facto standards of great utility. This approach is also useful because it increases the value of a translator. Translating a persistent representation into a canonical representation can make the ultimate interface available for that persistent data. Up to this point however, all attempts to resolve the problem of data inter-operability using the ultimate interface approach have foundered on the limitations of the proposed interface, or upon non-technical constraints or characteristics of the proposed interface. Sometimes the ultimate interface approach turns out really to be a Trojan Horse designed to advance yet another ultimate representation approach.

So what is the real solution to the data inter-operability Tower of Babel? And why with all the inventiveness of the human race has the computer industry not thrown up such a solution in the past fifty years? We believe that the solution is neither an ultimate persistent representation for stored data, nor an ultimate interface for manipulating stored data. Both of these solutions are inherently static, and turn their face against entropy so that ultimately all they can do is build the data inter-operability Tower of Babel higher. Instead the solution is a representation independent Broker that is dynamically capable of supporting a variety of different interfaces against an even wider variety of persistent data representations. In theory a Broker is the universal interpreter caricatured by Lucas' 3CPIO, able not only to translate stored data between an almost infinite number of persistent representations, but also to infer and superimpose the necessary semantics and gestures. In practice a Broker provides an approach which can be used to define a product architecture where human ingenuity can be brought to bear on the problem of engineering usable solutions to theoretically intractable problems. Whatever the actual size of the data inter-operability Tower of Babel, a Broker can make it appear manageable.

We call this approach to solving the problem of data inter-operability the Universal Data Access Broker. Yet at first sight the Universal Data Access Broker still appears to be an unbounded engineering problem. Mathematically there are potentially even more interfaces to persistent representations for stored data than there are representations.

So how can a Universal Data Access Broker be produced? The answer lies in the fact that whilst there are an infinite variety of interfaces to persistent data representations, there are increasingly a bounded number of interfaces which be used "most of the time." This reflects the counter-balance to the human tendency to expand the data inter-operability Tower of Babel. This counter-balance is the equally powerful human tendency to take the easiest path. We observed that increasingly the tools for manipulating stored data have converged on a set of interfaces that are used as common plumbing, or middleware. A compelling example of this is the widespread use and abuse of Microsoft's ODBC [Gei94] interface. Middleware interfaces like ODBC that are required by widely used tools are often supported even if the competitive advantage of a particular persistent data is at odds with the interface. In the case of ODBC implementations of the interface are available even for database technology like object oriented databases [CBB97], which are in other ways fundamentally competing with the relational underpinnings of ODBC [SQL92]. The value of the middleware interface has transcended the contradictions and difficulties of applying the interface to the persistent data representation. This phenomenon, most likely an emerging paradigm shift, means that by both intent and in other cases accident certain middleware has become fundamental as the glue that binds together the tapestry of tools which we now use to develop systems. By leveraging this paradigm shift a Universal Data Access Broker can be produced which resolves a huge part of the data inter-operability Tower of Babel by supporting middleware interfaces.

3 What is Data Inter-Operability? Revisited

Before discussing the Universal Data Access Broker in more detail we believe it is important to distinguish between:

- *Syntactic Data Interoperability*. Implemented by a translator. Data is simply transformed from one persistent representation to another, typically in discrete (or batch) operations.
- *Semantic Data Interoperability*. This transforms data from one persistent representation into an implementation of an interface. This requires both transformation of the data into new representation and that "something" implements the semantic behavior corresponding to the new persistent form of the data. Rather than being transformed, data is "published" using a specific interface.

Syntactic data inter-operability can be enormously useful. Even so a typical phenomenon is that as soon as the user can access transformed data they will discover that they need to do more than "just" access it. In a typical business domain (where relational technology is pervasive) they may want to query it, update it in place and support all of the semantics for the data which will allow tools to be used. With syntactic data inter- operability this typically requires the use of syntactic inter-operability to transform the data into a canonical representation in a fixed persistent data form which can supply the missing semantics. For example the transformed data may actually be loaded into a relational database. Unfortunately this raises a series of complex new problems, of which the most obvious is the often intractable problem of synchronizing multiple representations of the same data.

Semantic data inter-operability is fundamentally more difficult because as part of the transformation process "something" must supply the semantics. Also semantic data inter-operability typically requires that data be transformed in ad-hoc ways, corresponding to the desired interface (often referred to as *on-the-fly* transformation). Whether the semantics can be inferred, or require human intervention to supply "hints" or even complete implementations depends on the mapping between the persistent data representation and the desired interface. For semantic data inter-operability to be useful we expect it to have to support a number of interfaces which the user will require. We further distinguish semantic data inter-operability based on exactly how those interfaces are supported resulting in two different kinds of semantic data inter-operability:

- Shallow – where the semantics of the desired interface are implemented against the persistent representation of the data.
- Deep – where the semantics of the desired interface are maintained inside a more complex weave of semantics. For example deep semantic data inter-operability allows for the implementation of a desired interface inside a transaction context which requires two-phase commit, or inside a secure context which requires that all communication be encrypted. These additional characteristics are not defined by the desired interface, but by other interfaces in the infrastructure. In effect the Broker is implementing not just one but a collection of interfaces against the persistent data representation:

A Universal Data Access Broker is capable of supporting both shallow and deep semantic data inter-operability. In fact it is likely that Brokers will support both kinds of semantic data inter-operability in order to allow users to make cost/benefit decisions. This is particularly true since we observe that achieving semantic data inter-operability is often described in other contexts as "legacy integration," or "legacy wrapping" and is a universal recurring problem which up to now has typically been solved by ad-hoc means. The Universal Data Access Broker provides a consistent approach and underpinning for attaining semantic data inter-operability in a wide variety of different domains.

4 Implementing a Universal Data Access Broker

The requirement for semantic data inter-operability is hardly new. The Broker approach is not obscure and consequently the construction of these kind of Universal Data Access Brokers has been attempted in the past. But it has always fallen short of the goal. Not so many years ago, if someone had proposed building a product which could support a standard API for data access which could be used from any platform they would have been considered deluded. So why can a Universal Data Access Broker finally be built now?

We believe this is because the appropriate technological underpinnings for a Universal Data Access Broker are finally available. We believe that the failure of previous attempts at constructing a Universal Data Access Broker can be traced not to inherent flaws in the approach, but instead to the lack of those underpinnings and the technological and economic barriers that this presented to those attempts. The fact that it took so long for these technological underpinnings to become available is at least in part because these underpinnings represent very fundamental paradigm shifts in the way that we approach the development of systems which manipulate persistent data.

We believe that the missing technological underpinnings for Universal Data Access Brokers were:

- Widely available and economically exploitable multi-threading and multi-processor systems.
- Universally available and economically exploitable infrastructures for distributed computing.
- The availability of Java, and Java based interfaces for data access.
- Widespread use and acceptance of object oriented approaches to building systems.

Let's address each of these in turn in terms of their impact on constructing a Universal Data Access Broker:

Multi-threading. Multi-threading is the ability of a single program (or process) context to establish a number of different execution contexts and allow the operating system to map those execution contexts onto a number of available processors. Only relatively recently has useful multi-threading become economically available on classes of systems other than mainframes. Multi-threading as a theoretical technique is well known and described [Kle95, Lea97], but without widespread availability (and corresponding cost) of Symmetric Multi-Processor (SMP) systems the benefits of using multi-threading were constrained. We believe that widely available multi-threading has the impact of a paradigm shift. This is because it allows the use of design and implementation techniques for massive scale-ability that had previously only been available to developers on mainframe class systems. For the purposes of a Broker multi-threading allows the construction of a Broker that scales to the necessary user configurations, sometimes thousands or tens of thousands of clients. Multi-threading also allows a Broker to provide effective returns on investment as more processors are added to an SMP system on which it runs.

Universally available distributed infrastructures. When we talk about data inter-operability we implicitly assume that there is some means of accessing a persistent data representation from the context in which we want to manipulate that data. Yet in the past the assumption of even this weak kind of inter-operability was impossible. Different types of systems were incompatible islands that could only be persuaded to talk to each other using low level network programming. Not long ago it would have been almost lunatic to conceive of an infrastructure that would allow programs on any system, in almost any language to inter-operate. Yet with CORBA 2.0 [CORBA], this is now a reality. This observation reinforces the fact that only part of the value of CORBA lies in the "distributed object" capabilities. Much of the key value of CORBA lies in the fact that it is practical to assume that interaction with a CORBA based system will always be possible at reasonable cost, no matter what platform or language the interaction is required in. This allows the construction of Brokers that do not have to solve the impossible combinatoric problem of supporting basic cross platform and cross language inter-operability. A universal distributed infrastructure is absolutely required for a practical Broker, not only to support appropriate de-facto drivers on each platform, but also to allow development in absolutely any desired configuration.

The availability of Java, and Java based interfaces for data access. Java as a technology is extensively documented elsewhere [GJS96] and does not need description here. The importance of Java to a Broker lies in the standard interfaces which JavaSoft have defined based on Java, in particular the JDBC interface for manipulating persistent data [HCF97]. JDBC is a Java based object oriented wrapper for a relational API, which has some functional similarities to ODBC. Whatever the imperfections of the first version of JDBC, for the purposes of a Broker in combination with Java it provides instant, out of the box connectivity from a client on any target platform to any persistent data representation supported by a Broker. Supporting JDBC is different from supporting CORBA. CORBA provides more capabilities than JDBC but requires development on the target platform using CORBA tools. Java and JDBC provide a universal interface that eliminates all barriers to initial use of a Broker.

Widespread use and acceptance of object oriented distributed systems. Perhaps the Internet should be entirely separate from the problem of data inter-operability. Yet in practice is it not. The Internet has forever changed the playing field on which we must deploy systems. Many systems must now meet scale-ability requirements that previously only a small number of systems had to contend with. Not only that but almost every major corporation has been plunged into a competitive environment in which the deployment of these systems is critical. The search for solutions to building highly scalable systems has accelerated the acceptance and deployment of object oriented distributed systems. In an object oriented distributed system it is typical for specialized services to be delegated by business objects to discrete implementations, which will provide critical capabilities such as data access, fault-tolerance, load balancing and so forth. This is in sharp contrast to the way in which earlier distributed systems were built. Earlier distributed systems were more monolithic and essentially required Broker technology, if used, to be platform specific and embeddable. A Broker solution to data inter-operability fits perfectly with the object oriented architectures of these distributed systems. In addition a Broker solution has the ability to scale when other solutions to data inter-operability cannot.

5 DataBroker – A Universal Data Access Broker

Identifying these paradigm shifts in technological underpinnings motivated us to design and implement a Universal Data Access Broker, DataBroker, which is a practical solution to the problem of data inter-operability.

For practical and economic reasons the shipping versions of DataBroker are constrained in terms of the interfaces and the persistent data representations that are supported. For economic reasons we chose to prove the viability of the Broker concept by producing a Broker based implementation of the de-facto standard JDBC interface for Java. Again for economic reasons we chose initially to support persistent representations which were in widespread use and for which there was demand for a high quality JDBC driver. These included both relational databases and mainframe persistent data representations such as DB2 and IMS. However neither of these choices precludes the subsequent use of DataBroker to support other interfaces, or to implement those interfaces against less widely used persistent data representations. In fact I-Kinetics has already experimented both with the development of other interfaces and the development of prototype tools such as a “Batch Plugin” which allow almost any persistent data representation to be “Brokered” into the interfaces supported by DataBroker.

From our experiences with DataBroker we believe that there are several key lessons:

- DataBroker both confirms and leverages the fundamental paradigm shifts we had identified, as outlined in more detail below. In particular we believe we have fully vindicated our belief that CORBA was the perfect choice as an infrastructure for a Broker because of the fundamental emphasis on platform and language inter-operability.

- DataBroker demonstrates that a Broker approach will allow the practical and relatively rapid construction of support for specific interfaces against almost any kind of persistent data representation.
- DataBroker vindicates our belief that not only is a Broker based architecture viable for high performance manipulation of persistent data, in fact a Broker based architecture is able to support the construction not just of competitive implementations of interfaces but of leading implementations. This perhaps deserves a longer discussion, since it reflects on a wider debate on whether two tier or n-tier architectures are more effective.

DataBroker was designed to support data inter-operability, which meets the following practical requirements:

- Robust. A Broker must be capable of 7x24 operation.
- High performance. A Broker must both deliver high performance to an individual client and sustain that performance when hundreds, thousands and perhaps tens of thousands of clients are being supported.
- Scalable. A Broker must be able to meet the demands of deployment on a development Intranet, with a few users, and a production Internet with hundreds, thousands or even tens of thousands of users.
- Adaptable. A Broker must be able to deal with data in predictable formats, like relational database tables, as well as data in less friendly formats, in both files (whether application specific or generic) and non-relational databases.
- Secure. A Broker must be able to provide appropriate security to meet specific business objectives at a cost that matches the benefit and without imposing unwanted overheads on all usage of the Broker.
- Manageable. A Broker must be capable of being deployed and managed across a widely distributed system, thus it should be manageable (in appropriate circumstances) across the Internet and a Broker must integrate with existing standards for managing distributed systems such as SNMP.
- Standard. A Broker must follow both formal and de-facto standards in a way which maximizes the return on investment for users and which as far as possible allows the user to make decisions about technology and tools which are not solely based on the constraints of the underlying persistent data representation.

DataBroker meets these requirements by a combination of high quality engineering and by the explicit leverage of the four paradigm shifts we have previously identified.

The impact of Multi-threading. DataBroker is implemented as a multi-threaded CORBA based server that allows the mapping of multi-threading onto the semantics of the desired interface. For example for JDBC DataBroker uses a session based threading strategy where each thread corresponds to a specific JDBC Connection which is being mapped to a session against the persistent data representation. In addition DataBroker uses multi-threading as an optimization technique at multiple levels. In the implementation of the interface, for example by separate threads for pre-fetch of data. In the implementation of the Broker, by separate threads for pre-initialization of required objects and overheads such as garbage collection. In interacting with the persistent data representation, for example by separate threads for pre-fetch of data and pre-initialization of required objects. In toto we have found that these optimization techniques helped us build a much faster implementation of JDBC than would be possible with a non multi-threaded approach.

The impact of using CORBA. The use of CORBA throughout DataBroker is pervasive. Internally DataBroker uses only CORBA IDL to define system interfaces and IIOP as the communication protocol. Because of this DataBroker integrates seamlessly with capabilities such as Security, Naming and Transactions provided by our Orb vendor [OTM97]. In addition the use of CORBA has allowed us to continuously develop and refine the internal interfaces which we use in the search for performance. Because of this continuous refinement DataBroker now defines an internal protocol which is tightly optimized for the support of interfaces such as OPENjdbc. This allows the combination of DataBroker and OPENjdbc to exceed the performance of JDBC drivers that are implemented using a simple two-tier approach against general-purpose relational database interfaces. This violates conventional wisdom, that introducing an extra tier into a distributed system will slow down performance. In fact using a Broker can actually improve performance.

The impact of Java & JDBC. Initially DataBroker supports both JDBC and a CORBA IDL interface. For pragmatic reasons the initial CORBA IDL interface is itself based on JDBC functionality, with modifications to support usage from "thin" CORBA clients. Overall JDBC has had a pervasive impact on DataBroker. In some ways the first version of JDBC (JDBC 1.22) still presents a low-level object oriented interface to persistent data. But the limitations of the first version of JDBC have been compensated for by the enormous acceptance of the interface by tool vendors and the consequent rapid maturity of the market. For the most part the implementation of the JDBC interface against DataBroker currently supports shallow semantic data inter-operability. We anticipate that JDBC 2.0 will have an equally significant impact on DataBroker in a different way, by emphasizing the importance of a Broker to support multiple interfaces for deep semantic data inter-operability.

Developing DataBroker as a CORBA based component. We chose to develop a Broker that has a component based architecture, anticipating deployment in multi-platform distributed systems. The result of this is that DataBroker can be engineered using the CORBA Orb to provide capabilities such as distributed management and "thin" clients. This also allows us to allow for deployment of DataBroker capabilities as discrete components (plug-ins).

6 A Universal Data Access Broker in Use

In these two examples the problems and system architectures are real, but the company names have been left out.

The first example is a credit card transaction system. This system takes in credit card transactions from various vendors on the Internet and processes them in a central relational database. Volume is high and 7x24 availability is fundamental. The transaction implementation uses JDBC and SQL directly and is functionally simple. A credit card transaction request is fielded, passed to a processing database which checks and flags it and finally a transaction approved or failed response is issued. However the architecture of the system cannot be simple because of the volume of transactions which it must support, there may be as many as 1000 concurrent users. This imposes a high processing load on all of the system components, the web server, the Java JVM and the database. During deployment of the initial version of the system both performance and reliability were unacceptable, the system was taking perhaps ten minutes to respond to incoming transaction requests, connections to the system were failing and the system would crash under stress. None of these problems had been present or obvious from prototype testing.

The system architecture used a single host on which the web server, Java JVM and database were all located. A simple type-4 JDBC driver was used to transmit SQL based operations to the database. Profiling tests demonstrated that:

- The web server, JDBC driver and database server together were consuming up to 6MB of virtual memory for each connection.

- Connections could become blocked in the Java JVM, continuing to consume virtual memory even though the connection was dead.
- The web server was a single point of failure.

Based on these results and other analysis the system architecture was changed to a multi-tier architecture so that:

- The web server would be on a separate dedicated host.
- The database server would be on a separate dedicated host.
- There would be a number of separate hosts each of which would support an instance of a Broker, the JDBC driver for the Broker, the Java JVM and the actual transaction implementation. The initial number of hosts selected was two, but this could be scaled upwards as necessary.

This new system architecture also allowed the Broker to offload connection management from the web server and database server. Because the Broker uses a multi-threaded architecture the virtual memory requirements for each connection dropped dramatically. In addition connections were load balanced from the web server across the hosts running an instance of the Broker. The Broker also supplied a fail-over capability which meant that an attempt to connect to a Broker instance which was somehow unavailable would be automatically routed to a different Broker instance. Dropping the Broker into the architecture did not just replace the JDBC driver with a faster implementation. It also filled in a number of key requirements for scalability which had previously been missing without modifying the transaction implementation.

The second example is a claim processing system. This system already uses a three-tier approach. The middle tier implements a web server and the Java based business logic. Another tier supports the database against which the business logic is implemented using JDBC. Finally the clients represent the third tier, interacting with the business logic tier via HTML and the web server. The system is used by both claims processing agents and 3rd parties, at any one time there may be as many as 3000 users of the system. To support this the development team implemented the above architecture using a high performance Java servlet that exploited multi-threading, connection pooling and caching of data in the business logic. They were confident that the three-tier architecture in combination with this sophisticated implementation of the servlet would scale to meet the requirements.

But it did not scale. The cause was hidden inside the sophisticated servlet. The developers had used a simple type-4 JDBC driver to implement the SQL calls against the database. This turned out to be the weak link in the system architecture. Despite the sophisticated approach used for the servlet the performance of the deployed system foundered because:

- The driver did not stream and cache data efficiently and could not deliver the necessary performance under load.
- The driver did not manage connections efficiently and over time would eventually cause the server to fail.
- The driver was thread-safe, but not multi-threaded. It was not capable of exploiting resources in the same way as the servlet.

To resolve this problem the development team added a Broker and the JDBC driver for the Broker to the middle tier of the system architecture. The multi-threaded architecture of the Broker reinforced the sophisticated implementation of the servlet, rather than adding a weak link to the chain of components that had to scale to meet the requirements.

7 Is a Universal Data Access Broker a viable solution to the problem of data inter-operability?

We believe the answer is an emphatic yes. We believe DataBroker demonstrates the viability of the Broker approach as well as less conventionally accepted conclusions, such as the ability of the Broker approach to deliver the highest performance for the interfaces that it supports.

Of course there will still be problem domains where specific requirements mandate the use of a specific persistent data representation with appropriate characteristics, such as an object oriented database in one case or a relational database in others. However whichever technology is selected for the development of a system in a specific problem domain the problem of data inter-operability with systems in other domains remains. It is guaranteed that these systems will not use the same persistent data representation. We believe that a representation independent Broker represents the only viable approach to delivering practical and usable data inter-operability which will scale to the levels required by the next generation of distributed systems.

References

[CBB97] Cattell, Barry, and Bartels. The Object Database Standard. ODMG 2.0. Morgan Kaufmann, 1997.

[GJS96] Gosling, Joy, and Steele. The Java Language Specification. Addison Wesley, 1996.

[Gei94] K. Geiger. Inside ODBC. Microsoft Press, 1994.

[HCF97] Hamilton, Cattell, and Fisher. JDBC Database Access with Java. Addison Wesley, 1997.

[Kle95] Kleiman. Programming with Threads. Prentice Hall, 1995.

[Lea97] Lea. Concurrent Programming in Java : Design Principles and Patterns. Addison Wesley 1997.

[OLE97] Microsoft OLE DB 1.1 Programmers Reference and Software Development Kit. Microsoft Press, 1997.

[OMG92] Object Management Group. The Common Object Request Broker: Architecture and Specification. 1992.

[OTM97] Orbix OTM Users Guide, IONA, 1997.

[SQL92] Data Management: Structured Query Language (SQL), Version 2. The Open Group, 1996.

Exporting Database Functionality — The CONCERT Way

Lukas Relly Heiko Schuldt Hans-J. Schek
Database Research Group, Institute of Information Systems, ETH Zürich, Switzerland

Abstract

In order to extend database technology beyond traditional applications a new paradigm called "exporting database functionality" as a radical departure from traditional thinking has been proposed in research and development. Traditionally, all data is loaded into and owned by the database, whereas according to the new paradigm data may reside outside the database in external repositories or archives. Nevertheless, database functionality, such as query processing, indexing, and transaction management, is provided. In this paper we report on the CONCERT *project that exports physical database design for advanced applications, and we discuss the consequences for transaction management, that becomes an important coordination task in* CONCERT.

1 Introduction

Todays Database Management Systems (DBMS) make the implicit assumption that their services are provided only to data stored inside the database. All data has to be imported into and being "owned" by the DBMS in a format determined by the DBMS. *Traditional database applications* such as banking usually meet this assumption. These applications are well supported by the DBMS data model, its query and data manipulation language and its transaction management. *Advanced applications* such as GIS, CAD, PPC, or document management systems however differ in many respects from traditional database applications. Individual operations in these applications are much more complex and not easily expressible in existing query languages. Powerful specialized systems, tools and algorithms exist for a large variety of tasks in every field of advanced applications requiring data to be available in proprietary or data exchange formats.

Because of the increasing importance of advanced applications, DBMS developers have implemented better support in their systems for a broader range of applications. Binary Large Objects provide a kind of low-level access to data and allow individual data objects to become almost unlimited in size. Instead of storing large data objects in BLOB's, some newer systems such as ORACLE (Version 8) and Informix (Dynamic Server with Universal Data Option) provide the BLOB interface also to regular operating system files. Because the large objects in any of these two options are uninterpreted, database functionality for this kind of data is only very limited. In order to better support advanced applications, the standardization effort of SQL3 specifies, among others, new data types and new type constructors. Most recently, SQL3 and object-orientation have fostered the development of generic extensions called datablades [10], cartridges [14], and extenders [9]. They are based on the concept of abstract data types and often come with specialized indexing.

Although they provide better support for advanced applications, however, except for the file system case, they all have the same fundamental deficiencies: Firstly, it is the DBMS together with its added extensions that prescribes the data structure and data format of the data to be managed. The consequence is that all complex specialized application systems and tools must be rewritten using the data structures enforced by the DBMS, or at least complex transformations must take place to map the DBMS representation into the application representation. Secondly, the DBMS owns the data. All data has to be physically stored inside the DBMS requiring to possibly load gigabytes of data into the database store.

These observations led to a radical departure from traditional thinking as it is expressed in [24]. In the COSMOS project at ETH [19, 20, 21, 13, 23], we focus on *exporting database functionality* by making it available to advanced applications instead of requiring the applications to be brought to the DBMS. Figure 1(a) shows the traditional DBMS that offers its functionality to clients whose data are under complete control of the DBMS. As opposed to this, figure 1(b) shows the new paradigm we have followed: The DBMS changes its role and becomes a coordinator (DBCoord) of many local specialized component systems. DBCoord provides database access to external data sources stored in the component systems. It provides tools for creating indexes over external data as well as for replications of external data. Exporting database access to external data sources does not exclude clients from accessing this data directly. In contrary, we explicitly want to deal with given specialized application systems and we do not require existing applications to be rewritten. We additionally want to provide DB functionality for (new) applications that need it. DBCoord ensures that changes of external data are properly reflected in related derived indexes and replications, more generally, that dependencies between component systems are transactionally maintained. The DBCoord's task is to coordinate possibly heterogeneous, possibly distributed, possibly autonomous subsystems rather than to store and to own data.

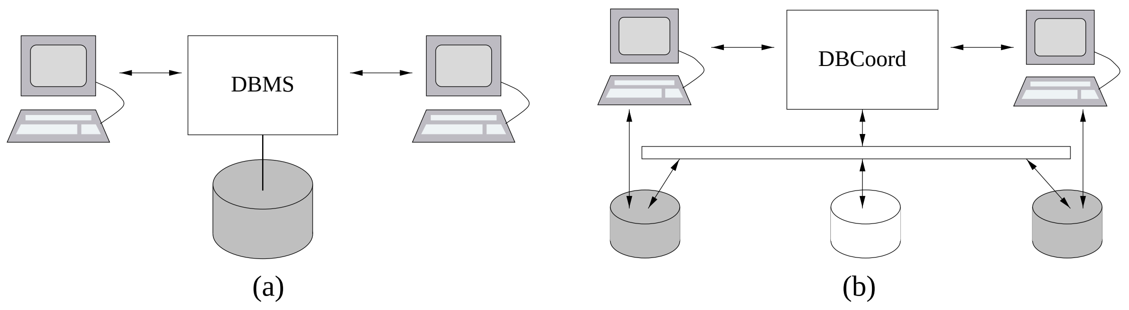

Figure 1: DBMS owning the data vs. DBMS providing database functionality

In this paper, we report on our research project COSMOS in general and the prototype DBMS CONCERT in particular [4, 16]. CONCERT provides implementations as "proof of concept". Here we discuss in more detail the aspect of physical database design and its consequences to transaction management. The paper is organized as follows: Section 2 introduces a sample application and focuses on the key concepts required for physical database design to efficiently provide access to external data. Then, in section 3, we discuss the implications to transaction management.

2 Exporting Physical Design: The CONCERT Approach

Throughout the paper, we will use a sample application — the management of geospatial images provided by a satellite — on which we show, how the concepts of providing database functionality for external data can be applied. A satellite is supposed to periodically generate geospatial images together with descriptive information (as, for instance, date and time the image has been made, the position of the satellite, etc). This information is stored in huge tape archives. Additionally, after an image has been taken, meta data describing this image is stored in a proprietary file format [7]. This meta data contains the descriptive information provided by the satellite and additional descriptions provided by a user. Furthermore, a preview of the geospatial image is materialized. The data objects to be managed therefore consist of the images in the tape archive and the meta data files (figure 2).

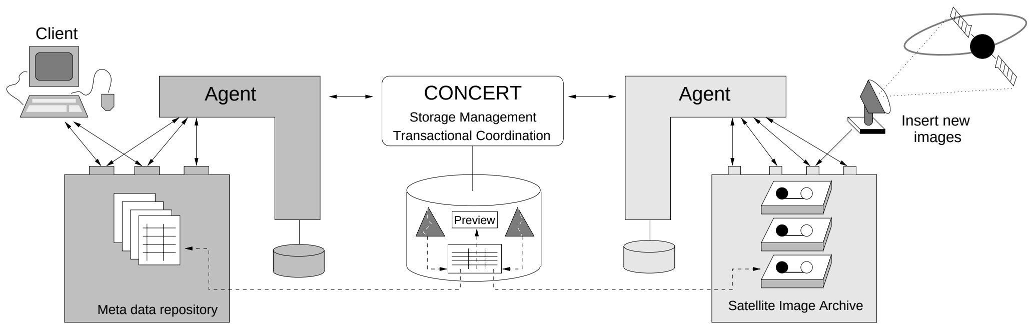

Figure 2: Management of satellite images and meta data stored at two different external repositories

In order to provide efficient access to the satellite image database, among others, the following physical design decisions can be considered.

- The relationship between the meta data and the image data has to be maintained. They should be viewed as the two sides of the same coin, although they are stored in independent systems.
- Index support for meta data attributes such as date and time of data capture should be available.
- Information retrieval techniques might be required to efficiently find images based on their description.
- Spatial indexing is required to access image data for certain regions.
- Sophisticated caching strategies for satellite images loaded off the tape archive have to be used.

Blades, cartridges, or alike allow the DBMS to be extended by *application specific types* and *access methods*. While implementing new types is relatively easy, new access methods is not. The new access method has to cooperate with the various components of the DBMS, such as concurrency control, data allocation, query evaluation, and optimization. This requires substantial knowledge of the DBMS internals. In contrast, CONCERT offers a limited, built-in set of physical design mechanisms in form of generic, trusted DBMS code provided by the DBMS implementor. Physical design is performed through relating new types to the fundamental concepts of the built-in physical design mechanisms.

In [25], Stonebraker introduced the idea of a generic B-Tree that depends only on the existence of an ordering operation to index arbitrary data objects. Our CONCERT approach generalizes this idea by identifying *all relevant concepts of physical database design* and expressing them by the so called *concept typical* operations required to implement them over external data. The data objects are treated as abstract data types (ADT) in CONCERT, and physical database design is performed based on the operations of the ADT only. In order to implement search tree access methods, a generic search tree approach (similar to GiST [8]) can be used as it integrates nicely into the CONCERT framework.

CONCERT provides the DBMS kernel functionality such as storage management, low level transaction management, and basic query processing. It's role is comparable to the one of RSS in System R. In order to enable good concurrency, CONCERT internally uses a multilevel transaction management scheme that is combined with ideas from [11, 6]. It uses a memory-mapped buffer on a multi-block page management [3] for efficient data access. It is beyond the scope of this paper to give full detail of the CONCERT approach. The following is a simplified sketch to give an idea. More detailed information can be found in [3, 4, 16].

Fundamental generic operations: the GENERIC concept In order to handle abstract data objects, to move them around in the DBMS and between remote repositories and to pass them between function invocations, the following three fundamental generic operations are required. In order to allocate space, the object's linearized **size** is required. The **copy** operation actually moves the object from one place to the other possibly performing necessary format transformations. During the copy operation, auxiliary resources (memory, file handles, network connections) might be required that are freed using the **free_aux** operation. These three operations are required for all CONCERT objects in addition to the concept typical operations that are discussed below.

The SCALAR concept The first specific concept is the concept of ordering, grouping, and clustering. It contains the generic B-Tree as a special case and is based on a comparison operation. The COMPARE operation of the SCALAR ADT has the following signature. It returns 0 in case of equality, otherwise -1 or 1 depending on the resulting order.

`COMPARE` (SCALAR_object, SCALAR_object) $\rightarrow \{-1, 0, 1\}$

This operation is sufficient to build a generic B-Tree (using techniques as in [25]) or any other access structure relying on an ordering or comparison such as a partition index, one of the new methods of physical design in the ORACLE8 DBMS tailored for advanced applications. In addition, the SCALAR ADT provides an optional HASH operation that allows certain data object to be used in the context of hash indexes.

Because arbitrary functions are allowed to implement the compare operation, for example remote procedure calls (RPC) can be used to compare external data objects residing on different machines or repositories. Obviously, for efficiency reasons, RPC might not be the appropriate method for external data and the compare operation can decide to create local copies. Also note that CONCERT concepts do not make assumptions about type information. For a single user object of a given type, several different concepts can be declared in order to instantiate, for example, more than one B-Tree on different user attributes.

The RECORD concept A second concept of physical database design is the one concerned with components or parts of objects. It is used implicitly in ordinary indexes that usually are built over an attribute (i.e., a component) of a data object. More important, the RECORD concept is used explicitly in vertical partitions. The relevant concept typical operation of the RECORD concept is the one identifying components of objects. It has the following signature:

`SUB_OBJECT` (RECORD_object, component) $\rightarrow$ object_part

extracting the object part identified by the parameter "component". The resulting object_part is a generic CONCERT object, that might be specified further by other concepts. The RECORD concept is especially important for objects with large, but infrequently used parts when vertical partitioning is used. Examples are objects that contain multimedia data such as images, audio, or video data. Often, only the descriptive attributes of such data objects are accessed. Therefore they are stored separately from the large multimedia components.

The LIST concept Information retrieval and its physical design techniques such as inverted file indexes and signature methods is a good example of the third concept. It represents a characterization of objects in order to efficiently answer queries looking for objects containing certain features. The relevant operations are the ones providing access to individual features of an object as follows:

`OPEN` (LIST_object) $\rightarrow$ cursor
`FETCH` (cursor) $\rightarrow$ feature
`CLOSE` (cursor)

In the simplest case, the resulting feature will be further specified using the SCALAR concept in order to use a B-Tree or alike for the inverted file index. The LIST concept is not limited to traditional information retrieval. It can support any kind of query asking for objects containing certain features such as, for instance, frequency spectra in audio signals or features in digital images. In these cases, the corresponding concept typical operations are the feature extraction or frequency analyzing operations.

The SPATIAL concept The last concept is the one concerned with spatially extended objects. They appear in many different contexts. Time intervals are spatially extended objects in a single dimension, spatial objects such as line segments and polygons in the context of GIS systems usually appear in a 2D space, CAD objects are similar, but in 3D space, and some applications are concerned with multidimensional data objects. The concept typical operations of the SPATIAL concept are:

> `OVERLAPS` (SPATIAL_object, SPATIAL_object) $\rightarrow$ boolean
> `SPLIT` (SPATIAL_object) $\rightarrow$ { SPATIAL_object }
> `COMPOSE` ({ SPATIAL_object }) $\rightarrow$ SPATIAL_object
> `APPROX` ({ SPATIAL_object }) $\rightarrow$ SPATIAL_object

The predicate OVERLAPS checks for a nonempty intersection of two spatially extended objects, SPLIT partitions an object into a set of partial objects, COMPOSE recombines objects that have been split and APPROX implements an approximation such as a minimal bounding box over a set of objects. It is up to the developer of the SPATIAL concept to make these concept typical operations meaningful to the application and its types. This means that the n-dimensional range query $\Box(q, \text{object})$ appears equivalent to the same query over the composition of the overlapping parts,

$$\Box(q, \text{object}) \equiv \Box(q, \text{COMPOSE}(\{o | o \in \text{SPLIT}(\text{object}) \wedge \text{OVERLAPS}(o, q)\}))$$

and for arbitrary query objects q, if they overlap any object $o \in obj$, they also overlap the corresponding approximation

$$\forall q (\exists o \in obj (\text{OVERLAPS}(o, q)) \Rightarrow \text{OVERLAPS}(\text{APPROX}(obj), q))$$

While the first three concepts are only used for data objects, the SPATIAL concept is used also for objects representing the data space in index nodes of a spatial index such as the space covered by an R-Tree node. An index node is split, for example, performing a SPLIT operation on the spatial object associated with the index node. The resulting objects define the data space of the new index nodes. The data objects are then distributed according to the OVERLAPS operation. Depending on the index strategy, the data object might be split into a set of smaller objects first. If data objects are inserted into a node, the corresponding data space can be calculated using the APPROX operation on the objects in the node. On retrieval, data objects that have been split need to be recombined using the COMPOSE operation.

Physical Design using CONCERT **Concepts** Using the CONCERT concepts, it is possible to get an integrated view of the satellite image database. Each image together with its meta data is associated with the RECORD concept containing components such as *image_data*, *preview*, and *description*. Access to the different components is performed through the concept typical operation SUB_OBJECT. For example, in order to compute the preview image for a satellite image, the operation

> image_sub_object (image, preview) $\rightarrow$ preview_image

runs the corresponding program accessing the tape archive, reading the satellite image and feeding it through a preview computation process. The required index support for meta data can be achieved by implementing the concept typical operation COMPARE for each attribute that an index is to be built over, such as

image_date_compare (image1, image2) $\rightarrow \{-1, 0, 1\}$
image_title_compare (image1, image2) $\rightarrow \{-1, 0, 1\}$

In order to apply information retrieval techniques on the descriptive information, the LIST concept is used. Its concept typical operation FETCH extracts the index terms from the description accessing the meta data repository in the following way

image_desc_open (image) $\rightarrow$ image_desc_cursor
image_desc_fetch (image_desc_cursor) $\rightarrow$ image_index_term

These index terms can then be stored in an inverted file. The SPATIAL concept allows CONCERT to build a spatial index over the images in the database. Accessing the image data requires loading it from the tape archive to secondary storage. Using the SUB_OBJECT operation,

image_sub_object (image, image_data) $\rightarrow$ raster_image

the loader program can be made known to the database enabling it to view the load operation as a materialization of a computed attribute. Therefore, standard database design decisions for materialized views can be used for the caching of the loaded image.

It is important to notice that only those concept typical operations required for the actual physical design have to be implemented. Whenever a new physical design method is to be used, the corresponding operations are implemented and registered in CONCERT. This allows for incremental improvement of the physical design as required by the applications.

3 Exporting Transaction Management

In our sample scenario, dependencies between external meta data files and index structures managed within the CONCERT DBMS exist as well as dependencies between the single parts of data objects stored in the tape archive and in the meta data files. Although data is manipulated by applications not being aware of these dependencies, *coordination* must reestablish overall consistency when it has been violated by local operations on external repositories. As data is no longer completely under the responsibility of the DBMS, this coordination can no longer be achieved by the local DBMSs transaction management. We therefore additionally have to export transaction management to keep track of dependencies. In what follows, we will concentrate on the coordination architecture, the processes that have to be executed by the CONCERT coordinator, and the transactional execution guarantees that have to be provided for these processes.

Coordination Architecture To enforce dependencies between subsystems, the CONCERT system has to act as global coordinator. The subsystems to be coordinated may be different heterogeneous and distributed resource managers or, in the case data is only accessible and interpretable via specialized services or applications, the application systems itself [13, 23]. Although, for coordination purposes, transactional properties of subsystems are required, we do not expect all subsystems to be DBMSs. However, in order to provide database functionality, a *database coordination agent* is placed on top of each subsystem [23]. Thus, from the point of view of the CONCERT coordinator, the subsystem together with its agent can be considered as DBMS. The functionality to be provided by the subsystems and their agents includes the atomicity of service invocations, the compliance with orderings of service invocations and either the compensation of already committed services or their deferment by a two phase commit protocol. When a subsystem is not able to directly provide this functionality, its agent has to implement it. Coordination agents are similar to 2PC agents proposed for federated database management systems [26] but provide considerably more functions in cases where the subsystem does not provide DBMS functionality.

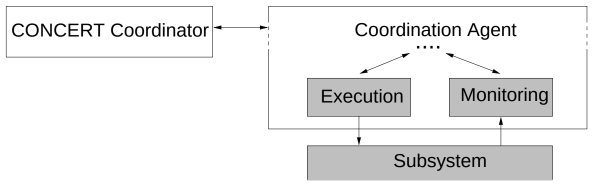

Figure 3: Subsystem specific part of generic coordination agent

Although certain parts of a coordination agent are tailored to the subsystem it belongs to, a generic structure of a coordination agent can be identified. Figure 3 shows the subsystem specific parts of such a generic coordination agent. When operations on external applications or repositories have to be invoked by the CONCERT coordinator, agents have to rely on existing services. Hence, the *execution module* of a coordination agent exploits, for instance, the API provided by a subsystem or performs a RPC call to a resource manager to be coordinated. All coordination agents together therefore overcome the inherent heterogeneity of the subsystems to be coordinated. As data within subsystems may be manipulated locally by local service invocations, information about these local operations has to be made available by the *monitoring module* of the respective coordination agent. Therefore, aside of heterogeneity and distribution, also autonomy of the systems to be coordinated is an important issue as both local transactions and global (coordination) transactions have to be considered [2, 12, 18]. Monitoring can, for instance, be performed on data level given the agent to be a DBMS itself by defining a database link and exploiting trigger mechanisms at the subsystem DBMS. However, in cases the subsystem is not based on a DBMS or if its data model is unknown, then monitoring has to be performed at application level, e.g., by plugging in trigger mechanisms at the application level. Often, systems like SAP R/3 [17] or Pro/Engineer [15] allow to pass control to the agent before some application function is executed.

Figure 2 shows the subsystems of our sample scenario, how they are provided with coordination agents and how the CONCERT coordinator communicates with external repositories and applications via these agents.

In addition to local transaction management in the subsystems and within CONCERT, we must add an additional transaction layer — the transactional CONCERT coordinator — keeping track of dependencies between external data. It should have become clear by now that the CONCERT coordinator provides functionality of an upper layer transaction manager coordinating subsystems while the agents and the CONCERT system provide local transaction management. Local transactions are intensively used for single operations. The agents, in turn, enrich a subsystem by a transaction manager at the application level enabling deferring or compensating single operations.

Coordination Processes in CONCERT A coordination process is a sequence of operations to be executed on subsystems [22]. For each coordination process, an execution guarantee is provided in the sense that at least one of several execution alternatives terminates successfully (generalized atomicity). The usual abort is treated as a special alternative. More generally, undoing everything is often not desired or even not supported by the subsystem and its agent. We will not present formal details (see [22]) but explain it using our example.

Considering our GIS application, coordination relies on appropriate coordination agents for both subsystems, the tape management and the meta data management. Each time a new satellite image is inserted into the tape archive locally, its agent notifies the CONCERT coordinator. In CONCERT, a record is then created representing the new image consisting in this state only of the *image_data* component (a reference to the image in the tape archive). Additionally, an index can be built over the date, the image has been taken by exploiting the concept typical operation *image_date_compare(image1, image2)*. Furthermore, a preview image is computed by *image_sub_object(image, preview)*, materialized in the CONCERT DBMS, and the image record is updated. The CONCERT coordinator then invokes the service *provide_meta_data* on the agent of the meta data repository. This agent requests a user for descriptive information of the new satellite image. After this information has been inserted in the meta data repository, the CONCERT coordinator is notified by the agent and the image record is updated (reference to the associated meta data description). Exploiting the concept typical operation *image_title_compare(image1, image2)*, an index is built over the title of the image provided by the user. Then, the index terms are extracted by the concept typical operation *image_desc_fetch(image_desc_cursor)* in order to store them in an inverted file. This coordination process is depicted in figure 4.

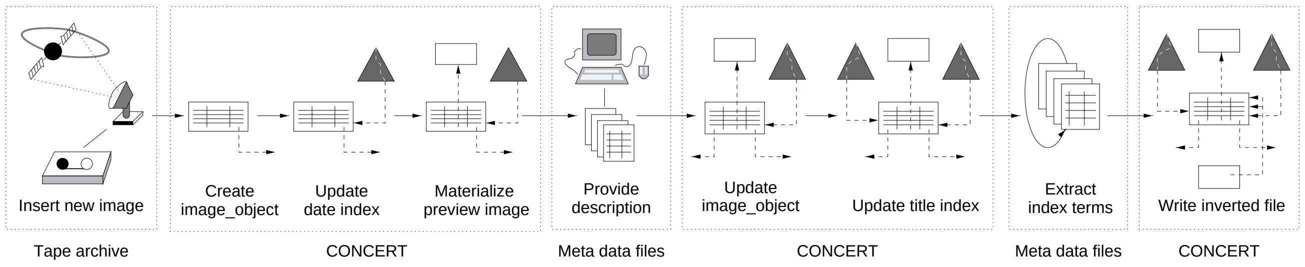

Figure 4: Coordination process to be executed when a new image is inserted into the tape archive

The CONCERT coordinator has to ensure that the execution of each coordination process leads to a system-wide consistent state even in case of failures and concurrency. With the generalized notion of atomicity for processes, at least one of the alternative executions (not depicted in figure 4) of our sample process has to terminate successfully. When, for instance, the date, the satellite image has been taken cannot be determined automatically, it is not desirable to undo all previous operations, e.g., delete the satellite image. It is more appropriate to alternatively ask a user for this information and continue executing the coordination process in order to ensure correct and guaranteed termination without undoing previous work. To enforce execution guarantees for processes, the global transaction management of the CONCERT coordinator relies on the local transaction management provided by the coordination agents. They have, for instance, to guarantee atomicity and isolation of local operations. Considering the local transaction within the meta data repository extracting index terms, the agent has to ensure that the sequence of *image_desc_fetch(image_desc_cursor)* operations is either executed successfully (thus returns all index terms) or fails and it has further to ensure that no changes of the meta data description are performed as long as the *image_desc_cursor* is open.

4 Conclusions

Exporting database functionality relaxes the assumption of traditional DBMSs to own all data to be managed and allows to offer this functionality to data that resides outside the DBMS. In the CONCERT project, we have identified four fundamental concepts of physical database design (SCALAR, LIST, RECORD, and SPATIAL) together with their concept typical operations. Physical design can be performed in an elegant way, based only on these concepts, for arbitrary data. By providing physical database design for external data, dependencies arise between data inside and data outside the CONCERT DBMS. We have shown, how the necessary coordination can take place in order to synchronize different data repositories with the help of subsystem-specific coordination agents. We showed, using a sample application, how both database services, physical design and transaction management, are provided by the CONCERT prototype system and how these services cooperate in order to contribute to a coherent whole. The aspects of physical database design and its consequences arising for transaction management, as we have discussed them, prove the feasibility of our approach.

In our current and future work, we generalize the ideas of transaction management to be exported in order to support coordination in two directions: Firstly, in the composite systems theory, we investigate arbitrary composite systems where — although scheduling takes place at different levels and schedulers can be connected in arbitrary ways — correctness of the whole system has to be provided [1]. Secondly, we decouple transactional properties in order to assign execution guarantees more flexibly to (sub)processes by exploiting the notion of spheres [5].

References

[1] G. Alonso, S. Blott, A. Fessler, and H.-J. Schek. Correctness and Parallelism in Composite Systems. In *Proceedings of the ACM SIGMOD/PODS Conference on Management of Data*, May 1997.

[2] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 2(1):181-239, Oct. 1992.

[3] S. Blott, H. Kaufmann, L. Relly, and H.-J. Schek. Buffering long externally-defined objects. In *Proc. of the Sixth Int. Workshop on Persistent Object Systems (POS6)*, pages 40–53, Tarascon, France, Sept. 1994.

[4] S. Blott, L. Relly, and H.-J. Schek. An open abstract-object storage system. In *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, June 1996.

[5] C. T. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17(2):179–198, 1978.

[6] G. Evangelidis. *The hB$^{\Pi}$-Tree: A Concurrent and Recoverable Multi-Attribute Index Structure*. PhD thesis, Northeastern University, June 1994.

[7] Content standard for digital geospatial metadata, June 8, 1994. Federal Geographic Data Committee (USGS/FGDC), U.S. Geological Survey, USA. `http://www.fgdc.gov`.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st Int. Conf. on Very Large Databases*, pages 562–573, Sept. 1995.

[9] IBM Corporation. `http://eyp.stllab.ibm.com/t3/`

[10] Informix Corporation. `http://www.informix.com/informix/bussol/iusdb/databld/datablade.htm`

[11] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, Dec. 1981.

[12] S. Mehrotra, H. Korth, and A. Silberschatz. Concurrency control in hierarchical multidatabase systems. *The VLDB Journal*, 6(2):152-172, May 1997.

[13] M. Norrie and M. Wunderli. Tool Agents in Coordinated Information Systems. *Information Systems*, 22(2):59-77, June 1997.

[14] Oracle Corporation. `http://www.oracle.com/st/cartridges/`

[15] Parametric Technology Corporation. `http://www.ptc.com/products/mech/proe/`

[16] L. Relly, H.-J. Schek, O. Henricsson, and S. Nebiker. Physical database design for raster images in concert. In *5th International Symposium on Spatial Databases (SSD'97)*, Berlin, Germany, July 1997.

[17] SAP AG, Walldorf, Germany. `http://www.sap.com`

[18] W. Schaad. *Transactions in heterogeneous federated database systems (in German)*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1996.

[19] H.-J. Schek, M. H. Scholl, and G. Weikum. From the KERNEL to the COSMOS: The database research group at ETH Zurich. TR 136, Information Systems, ETH Zürich, July 1990.

[20] H.-J. Schek and A. Wolf. Cooperation between autonomous operation services and object DBMS in a heterogeneous environment. In *IFIP, DS-5, Semantics of Interoperable Database Systems*, Australia, 1992.

[21] H.-J. Schek and A. Wolf. From extensible databases to interoperability between multiple databases and GIS applications. In *Advances in Spatial Databases: SSD'93*, LNCS. June 1993.

[22] H. Schuldt, G. Alonso, and H.-J. Schek. A unified theory of concurrency control and recovery for transactional processes. Technical Report, Swiss Federal Institute of Technology Zürich, 1998.

[23] H. Schuldt, H.-J. Schek, and M. Tresch. Coordination in CIM: Bringing Database Functionality to Application Systems. In *Proceedings of the 5th European Concurrent Engineering Conference*, Erlangen, Germany, April 1998.

[24] A. Silberschatz, S. Zdonik, et.al. Strategic directions in database systems – breaking out of the box. *ACM Computing Surveys*, 28(4):764–778, Dec. 1996.

[25] M. Stonebraker. Inclusion of new types in relational database systems. In *Proceedings of the International Conference on Data Engineering*, pages 262–269, Los Angeles, CA, Feb. 1986.

[26] A. Wolski and J. Veijalainen. 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous MDBMS. In *Proc. of the IEEE PARBASE'90 Conference*, pages 321–330, Miami, March 1990.

ACM SIGMOD/PODS ’99 Joint Conference
Wyndham Franklin Plaza**,** Philadelphia, PA
May 31—June 3, 1999

The ACM SIGMOD/PODS '99 Joint Conference will, once again, bring together the SIGMOD and PODS conferences under one umbrella. The two events will overlap for two days to form a joint conference, but each will have a separate third day without overlap. There will be only one registration process for the joint, four-day conference. SIGMOD and PODS will have separate program committees and will have separate proceedings.

Authors must submit their papers to the conference that they feel is most appropriate for their work. We recommend that applied papers be submitted to SIGMOD and theoretical ones to PODS. The same paper or different versions of the same paper should not be submitted to both conferences simultaneously. Attendees will receive both proceedings and be encouraged to attend sessions in both conferences. Some of the technical events and the lunches will be joint events.

Submission Guidelines

All research papers should be directed to the appropriate program chair.

Prof. Christos Faloutsos, (Re: SIGMOD ’99)
Computer Science Department
Carnegie Mellon University, Wean Hall,
5000 Forbes Ave, Pittsburgh, PA 15213-3891
Tel: (412) 268-1457, Fax: (412) 268-5576
Email: `christos@cs.cmu.edu`
SIGMOD Deadline: Nov. 2, 1998

Prof. Christos Papadimitriou, (Re: PODS `99)
Computer Science Division
Soda Hall, University of California
Berkeley, California 94720 USA
Tel: (510) 642-1559
Email: `christos@cs.berkeley.edu`
PODS Deadline: Nov. 16, 1998

The address, telephone number, FAX number, and electronic address of the contact author should be given on the title page of the submission. All authors of accepted papers will be expected to sign copyright release forms, and one author of each accepted paper will be expected to present the paper at the conference. Proceedings will be distributed at the conference, and will be subsequently available for purchase through the ACM.

SIGMOD Submissions: Original research papers are solicited (at most 25 pages excluding appendices, 1.5 spaced, in no smaller than 10 point font and with 1 inch margins). Submit your abstract electronically on or before 5pm EST, October 26, 1998. Submit six hard copies of your paper, to arrive on or before November 2, 1998, to the PC Chair. **VERY IMPORTANT** for scheduling possible patent filings: Authors of accepted papers will be expected to make their papers public by **March 23, 1999** (one week after the due date of the camera ready copies). Panel, Tutorial, Demonstration and Industrial Track proposals are also due on Nov. 2, 1998. They should be sent to the respective chairs. (Please see website below for further details.)

PODS Submissions: For hardcopy submissions, please send 16 copies of an extended abstract by the deadline of November 16, 1998 to the PC chair. Electronic submissions are encouraged. Please see the conference web site for further details. Submissions should be limited to 10 pages (with font size at least 11 pts). The abstract must provide sufficient detail to allow the program committee to assess its merits and should include appropriate references to and comparisons with the literature.
For further information, please see the Conference Web Site at **http: //www.research.att.com/conf/sigmod99**

First Call for Participation

ICDT '99

7th International Conference on Database Theory

Jerusalem, Israel, January 10-12, 1999

`http://www.cs.huji.ac.il/icdt99/`
`http://www.cis.upenn.edu/~icdt99/`

ICDT is a biennial international conference on theoretical aspects of databases and a forum for communicating research advances on the principles of database systems. Started in Rome 1986, it was merged in 1992 with the Symposium on Mathematical Fundamentals of Database Systems (MFDBS), started in Dresden in 1987. ICDT is considered one of the prestigious conferences in database theory.

The conference will take place in the Radison-Moria hotel in Jerusalem. It will feature a single thread of presentations, consisting of the papers selected by the program committee, invited talks, and one or more tutorials. Full details of the program, as well as hotel and registration forms, will be available at the home pages above no later than September 18, 1988.

The conference will be followed by a workshop on Query Processing for Semistructured Data and Non-standard Data Formats. For more details, see

http://www-rodin.inria.fr/external/ssd99/workshop.html
http://www.research.att.com/ suciu/workshop99-announcement.html

Important dates:

Registration starts:	**Sept 19, 1998**
Paper submission for the workshop	**Oct. 31, 1998**
Late registration starts	**Nov 22, 1998**
Conference:	**January 10-12, 1999**
Workshop:	**January 13, 1999**

Program Committee:

Catriel Beeri (co-chair), The Hebrew U.
Peter Buneman (co-chair), U. of Pennsylvania
Gustavo Alonso, ETH Zurich
Anthony Bonner, U. of Toronto
Marco Cadoli, U. di Roma "La Sapienza"
Sophie Cluet, INRIA
Guozhu Dong, The U. of Melbourne
Ronald Fagin, IBM Almaden Research Center
Erich Grädel, Aachen Technical U.
Gösta Grahne, Concordia U.
Marc Gyssens, U. of Limburg (LUC)
Yannis Ioannidis, U. of Athens (& U. of Wisconsin)
Alon Levy, U. of Washington
Alberto Mendelzon, U. of Toronto
Guido Moerkotte, U. of Mannheim
Shelly Qian, SecureSoft, Inc.
Kenneth Ross, Columbia U.
Timos Sellis, National Technical U. of Athens
Eric Simon, INRIA
Dan Suciu, AT&T Labs
S. Sudarshan, IIT Bombay
Michael Taitslin, Tver State U.

Program co-chairs:

Catriel Beeri
`beeri@cs.huji.ac.il`

Peter Buneman
`peter@cis.upenn.edu`

Local Organization: Tova Milo, Tel-Aviv University

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398