Extending relational database technology for new applications

by J. M. Cheng
N. M. Mattos
D. D. Chamberlin
L. G. DeMichiel

Relational database systems have been very successful in meeting the needs of today's commercial applications. However, emerging applications in disciplines such as engineering design are now generating new requirements for database functionality and performance. This paper describes a set of extensions to relational database technology, designed to meet the requirements of the new generation of applications. These extensions include a rich and extensible type subsystem that is tightly integrated into the Structured Query Language (SQL), a rules subsystem to enforce global database semantics, and a variety of performance enhancements. Many of the extensions described here have been prototyped at the IBM Database Technology Institute and in research projects at the IBM Almaden Research Center in order to demonstrate their feasibility and to validate their design. Furthermore, many of these extensions are now under consideration as part of the evolving American National Standards Institute/International Organization for Standardization (ANSI/ISO) standard for the SQL database language.

The development of relational database systems has been stimulated over the years by the rapidly growing demands of commercial applications. As a result, today's relational systems are oriented primarily toward commercial requirements, which typically include on-line transaction processing and decision support applications, based on simply structured data in tabular form. For this class of applications, relational database systems are a mature and robust technology.

Rapid reductions in hardware price and improvements in speed and capacity have led to the expansion of database systems into new application areas such as engineering design, multimedia, and medical systems. These applications often require the storage of data objects that are very large, semantically complex, or richly interconnected—requirements that relational database systems have not been designed to fulfill. Furthermore, the operations performed on these objects are likely to be much more sophisticated than those traditionally supported by relational database systems. For example, a computer designer might need to combine simple objects such as gates and storage cells into a complex object whose behavior is represented by a higher-level abstraction such as a shift register or arithmetic unit. Furthermore, the engineering design of a computer chip might be subject to some global constraint such as a limit on the total number of transistors.

Meeting the needs of this new class of applications is perhaps the greatest challenge facing today's relational database systems. We believe that this challenge can be met by efficiently extending relational technology with the following new features:

- A rich and extensible type subsystem that enables users to define their own data types and functions to encapsulate the semantic behavior of complex objects
- An efficient rule subsystem for protecting the global integrity of the database and providing active semantics for data
- Performance enhancements oriented toward the processing of large, complex, and richly interconnected data objects

We believe that these extensions will enable relational systems to meet the requirements of a new generation of applications, while retaining the traditional advantages of the relational approach. In this paper, we describe our view of such extended relational database management systems (DBMSs). Many of the extensions described here have been demonstrated in experimental prototypes in research and development laboratories within IBM. Some of them have also become available in commercial database products such as UniSQL**, INGRES**, Sybase**, and ORACLE**. IBM is playing an active role in the ANSI (American National Standards Institute) and ISO (International Organization for Standardization) database language committees that are defining the next version of the Structured Query Language (SQL) standard, currently known as SQL3.[1] Many of the extensions described in this paper are a part of this evolving standard.

Extensible type and function subsystems

One of the most important trends in database management is the trend toward increasing the semantic content of the data stored in the database. In object-oriented systems, this is accomplished by allowing users to define new data types (or "classes") and functions (or "methods") to represent the entities in their application domains. Relational database systems can be similarly extended to support a richer set of built-in and user-defined[1-6] types and functions, to store instances of these data types in tables, and to use these functions in queries. These facilities will enable the semantic behavior of stored objects to become an important resource that can be shared among applications.

Extending the type subsystem. Modern object-oriented languages support the important concepts of strong typing, user-defined types and methods, encapsulation, single and multiple inheritance, function overloading, and dynamic binding. All of these concepts can be incorporated into relational database systems while preserving the traditional advantages of the relational data model. In this section, we describe how strong typing, encapsulation, and inheritance can be supported by means of an extensible type subsystem. In a later section, we discuss how an extensible function subsystem can support function overloading and dynamic binding.

A database type subsystem can be extended both by providing additional predefined (built-in) data types and by providing a facility whereby users can define new data types. Data types that have well-known semantics and behavior, such as dates and times, can be most efficiently supported if they are made an integral part of the database system. In this paper we give examples of both new predefined types and facilities for supporting user-defined types.

Multimedia extensions. Multimedia applications are a very important part of the new generation of database applications. Recognizing the importance of this application category, ISO has recently begun a new project to study multimedia extensions to the SQL language, including extensions for spatial (involving lines, polygons, etc.) and temporal (time-varying) queries. Multimedia applications typically require the storage of very large but relatively unstructured objects such as images, drawings, and audio or video sequences. Such objects are sometimes referred to as binary large objects, or BLOBs. It is essential that an extended relational database system be able to store such very large unstructured objects as entries in tables. Techniques for efficient storage and manipulation of large objects have been investigated by the Starburst relational prototype project at the IBM Almaden Research Center.[7] Unlike most existing large-object implementations, the Starburst approach integrates BLOB support smoothly into the SQL language. This work has led to a proposal, recently approved by the ANSI, for incorporating BLOBs into SQL3.[8]

To implement BLOBs efficiently, it is important to minimize their movement from one place to another in storage. For this reason, the manipulation of BLOBs must be performed within the database as much as possible, with the final result being "delivered" to the application at the latest possible time. Indeed, many BLOB manipulations can be performed entirely within the database without ever copying the actual BLOB into the application space. This avoidance or delay in copying ("deferred materialization") of BLOBs is not only efficient, but also convenient because it frees the application developer from the allocation and manipulation of large buffers for holding intermediate results. In order to maintain good database clustering and to support the deferred materialization of BLOBs, a table entry for a BLOB should not contain the BLOB value itself, but rather a descriptor that indicates where the BLOB value is stored in a separate disk extent.

An important technique for the efficient manipulation of large objects, included in the IBM proposal to the ANSI SQL3 committee, is the concept of a handle.[8] A *handle* is an opaque token that represents the value of a BLOB and, as such, can be used in application programs. If a BLOB-expression is evaluated and assigned to a variable of type handle, the variable contains not the actual BLOB value, but a token that uniquely identifies this value. The handle indicates to the database system how the actual BLOB value can be assembled from fragments of one or more BLOBs that are stored in the database. The handle can be used in other expressions just as though it were an actual BLOB. In keeping with the value-based semantics of SQL, handles represent immutable values rather than references to large objects. Therefore, possible side effects that may occur with the use of references are eliminated. Handles cannot be used to modify the BLOB value they represent, but can be used to generate a new, modified BLOB value. A handle, once created, retains its validity until the end of the transaction in which it was created or until it is explicitly released. For example, the following statement, which uses the syntax of the SQL3 assignment statement, creates a new handle as the value of variable handle3:

```
SET :handle3 = concat(substr(:handle1, 0, 1000),
                      substr(:handle2, 0, 2000))
```

This handle uniquely identifies the BLOB generated by the right-hand side of the assignment. The execution of this assignment statement does not cause the movement of any actual data, and the new BLOB value is not actually retrieved (or "materialized") at this point. Instead, the handle contains the information required to assemble a new BLOB from the first 1000 bytes of the BLOB described by handle1 and the first 2000 bytes of the BLOB described by handle2. Of course, handle1 and handle2 may in turn contain information for assembling their BLOBs from fragments of other BLOBs stored in the database. The only time BLOB values are actually materialized is when they are assigned to a host variable that is a string buffer (not a handle), or when they are assigned to a column of a table.

A handle represents the value of binary large objects (BLOBs).

Since a handle is an opaque token, a language-dependent convention can be defined for declaring variables corresponding to handles in each host programming language. For example, handles might be declared in the C language by a statement such as:

```
HANDLE h1, h2, h3
```

that might in turn be translated by an SQL preprocessor into valid C declarations.

The following example illustrates how an application program might use a handle to represent a concatenation of two database BLOBs. In the example, the program retrieves small fragments of the concatenated BLOB, examines them, adds a third BLOB to the first two, and stores the result in a database table. This is accomplished with only a small buffer in the application program. The movement of the actual data is deferred as long as possible. Without handles, this example would be far more difficult to write and less efficient to execute. The example is given in abbreviated form, with the application program logic

omitted. The example makes use of new built-in functions defined on BLOBs (e.g., CONCAT, SUBSTR), treating them as strings of bytes.

```
/* In the host language, declare h1 and h2          */
/* to be BLOB handles, and buffer1 to be a buffer   */
/* of size 4000 bytes.                              */

SELECT CONCAT(resume, thesis) INTO :h1
FROM applicant WHERE id = :x;

/* So far, no data have been moved.                 */
/* The next statement materializes 4000 bytes of    */
/* the concatenated BLOB into the buffer.           */

SET :buffer1 = SUBSTR(:h1, 0, 4000);

/* After examining the buffer, we decide to retrieve */
/* another fragment of the concatenated BLOB.        */

SET :buffer1 = SUBSTR(:h1, 10500, 4000);

/* Having approved the applicant, we now store a    */
/* fragment of the resume and thesis, combined with */
/* a photo, into a table of successful candidates.  */

SELECT photo INTO :h2
FROM applicants WHERE id = :x;

INSERT INTO invitees
  VALUES (:x, CONCAT
            (SUBSTR(:h1, 0, 8000), :h2));

/* Now the selected fragments of the original BLOBs */
/* are materialized, concatenated, and copied into  */
/* the new table.                                   */
```

The handle concept described above is very important in meeting the performance requirements associated with the manipulation of very large objects and, at the same time, fits nicely into the semantics of the SQL language. Although it has been prototyped at the Almaden Research Center and recently accepted by ANSI for incorporation into SQL3, we believe that it cannot be found in existing relational DBMSs.

User-defined data types. An extensible type system should support several categories of user-defined types, including distinct types, abstract data types, and language types. Each of these type categories is described below. User-defined types can be supported by the addition of a new SQL statement, CREATE TYPE. The CREATE TYPE statement has several variations, each of which permits a different category of user-defined type to be constructed from one or more existing types.

•*Distinct types*—A distinct type is a user-defined data type that shares its internal representation with an existing type (its "source" type), but is considered to be a separate and incompatible type for semantic purposes. For example, one might want to define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use BLOB for their internal representation.

The following example illustrates the creation of a distinct type named audio:

```
CREATE DISTINCT TYPE audio AS blob;
```

Although audio has the same representation as the predefined data type BLOB, it is considered to be a separate type that is not comparable to a BLOB or to any other type. This allows the user to define functions written specifically for audio and to be sure that these functions will not be applied to any other type (pictures, text, etc.).

Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, since these may not be meaningful. (For example, the LENGTH function of the audio type might return the length of its object in seconds rather than in bytes.) However, a user can explicitly specify that certain functions and operators of the source type should also apply to the distinct type as described later in the section "Sourced Functions." Such functions and operators can be invoked with no additional run-time cost because the representation of the distinct type is the same as that of the source type.

•*Abstract data types*—An abstract data type (ADT) is a structured data type consisting of a sequence of heterogeneous named attributes, whose values may themselves be of any database type (including other ADTs.) The following statement illustrates the creation of an abstract data type named t_address:

```
CREATE TYPE t_address
    (number     INTEGER,
     street     CHAR(30),
     city       CHAR(20),
     state      CHAR(2),
     zip        CHAR(5));
```

An abstract data type (as well as any other user-defined type, e.g., a distinct type) can be used in the definition of a column of a table, as in the following example:

```
CREATE TABLE employee
    (id              INTEGER,
     name            CHAR(30),
     birthdate       DATE,
     work_address    T_ADDRESS,
     home_address    T_ADDRESS,
     picture         IMAGE);
     /* IMAGE is a distinct type on BLOB */
```

Instances of ADTs are *encapsulated* in that their attributes can be accessed or modified only by functions defined on the ADT. The creation of an ADT automatically generates accessor functions (to return the values of the attributes), mutator functions (to modify the values of the attributes), and a constructor function (to create a new instance of the ADT). The use of these functions can be restricted in order to protect the encapsulation of the ADT. An ADT can be used to represent a class of objects with complex internal state and behavior, implemented by means of user-defined functions that operate on the ADT, using the system-generated accessor, mutator, and constructor functions as primitives.

It is transparent to a user whether a function on an ADT is returning a stored attribute of the ADT or a computed value. Thus, the internal representation of an ADT can be changed without affecting running applications. Hence, in the above example, the definition of t_address could be changed in such a way that the zip attribute is not stored, but rather is computed by a function based on the attributes number, street, and city. Such a change would be transparent to applications that query this attribute using the zip accessor function, as in the following query:

```
SELECT zip (home_address)
FROM   employee
WHERE  id = 435423;
```

An important feature of object-oriented languages is *inheritance*, which promotes code reuse by allowing new types to be defined as specialized forms of existing types. An extended relational DBMS supports inheritance by allowing an ADT to be defined as a subtype of another ADT, its supertype. The subtype inherits all the attributes and behavior of the supertype, in addition to defining new attributes and behavior of its own. Inheritance is illustrated by the following example:

```
CREATE TYPE t_phone
    (area_code      INTEGER,
     phone_number INTEGER);

CREATE TYPE t_bus_phone
    UNDER t_phone
    (extension      INTEGER);

CREATE TYPE t_priv_address
    UNDER t_address;

CREATE TYPE t_bus_address
    UNDER t_address
    (bus_phone      t_bus_phone);
```

In this example, t_bus_phone is defined as a subtype of t_phone, thus acquiring all of the attributes of t_phone as well as the additional attribute extension, which is an integer. Similarly, the type t_bus_address is defined as a subtype of t_address, and acquires all of its attributes in addition to the attribute bus_phone of type t_bus_phone. Thus, if a table named customer contains a column named work_addr of type t_bus_address, the following query would be valid:

```
SELECT extension(bus_phone(work_addr))
FROM   customer
WHERE  id = 284693;
```

Most object-oriented languages provide support for multiple inheritance—meaning that an ADT can be defined as a subtype of two or more other ADTs. Multiple inheritance is also a feature supported by the draft SQL3 standard. In the case that an ADT is a subtype of two or more ADTs that have a common supertype, the attributes of that common supertype are inherited only once.

Because ADTs are encapsulated such that access to their attributes is supported only by means of functions, the access to these attributes is ruled

by the algorithm used to select the most appropriate function for a given function invocation. In general, this algorithm selects the most specific function that is applicable to the types of the operands.[2] Thus, a subtype attribute will override a supertype attribute of the same name.

Inheritance of behavior (or functions) is supported in the sense that any function that is defined on a supertype is applicable to instances of its subtypes. In other words, an instance of a subtype can be used in any context where an instance of a supertype is expected. This principle is called *substitutability*. For example, assume the existence of a function assess_land_value defined on type t_address and returning a decimal value. This function can also be applied to any instance of a subtype of t_address (i.e., t_bus_address and t_priv_address), as in the following example:

```
CREATE TABLE employee
      (id               INTEGER,
       name             CHAR(30),
       work_address     t_bus_address,
       home_address     t_priv_address,
       birthdate        DATE,
       picture          IMAGE);
       /* IMAGE is a distinct type on BLOB */

SELECT assess_land_value(work_address)
FROM   employee
WHERE assess_land_value(home_address)
                                  > 100000.00
```

It is important to note that the support of abstract data types, and especially their use in defining columns of tables, greatly extends the expressiveness and modeling power of relational database systems. However, this extension is by no means a departure from the relational data model, since it deliberately exploits the neutrality of the relational model toward data types (or domains).

• *Language types*—The purpose of language types is to provide an interface between the database and programming language type systems. A language type specifies a host-language representation into which a database type can be transformed if necessary. Each language type is the representation for a specific database type in a specific host programming language. Transforming an instance of a database type into a corresponding language type enables it to be operated on by library functions written in the host language. User-defined language types have been used in experimental IBM systems,[2] but are not currently included in the draft SQL3 standard or in existing relational systems.

Most relational database systems support a primitive language type for each of their built-in database types for each host language. For example, the database type INTEGER might correspond to the language type LONG in C, and to the language type S9(9) COMP-4 in COBOL. An application program can declare variables in the language type and use them as input and output variables in SQL statements, as in the following C example:

```
int x, y;
/* x and y are host variables of type int */

SELECT salary INTO :x
FROM emp  WHERE id = :y;
```

If an extended relational database system permits users to define their own database types, it is also important to allow them to define the corresponding language types to be used for input and output of these database types. For example, the following statement might declare that the C language type corresponding to the database type t_ phone defined above is a structure containing four integers, two of which are used as null indicators:

```
CREATE LANGUAGE TYPE l_phone FOR t_phone
                                  LANGUAGE C
DECLARATION
    'typedef struct
       {
       int   areaCode;
       int   areaIsNull;
       int   number;
       int   numberIsNull;
       } l_phone;
    ';
```

The two functions that convert an instance of a database type into an instance of a language type, and vice versa, must be provided by the definer of the language type.

Once a language type has been defined, a user might use it to declare input and output variables to be used in SQL statements in much the same way as the built-in types, as in the following example:

```
l_phone x;
/* x is a host variable of type l_phone */
int y;
/* y is a host variable of type int        */

SELECT bus_phone(work_addr) INTO :x
FROM emp WHERE id = :y;
```

Extending the function subsystem. The database function subsystem can be extended by providing new built-in functions or by allowing the user to define functions. In this paper, we concentrate on user-defined functions since they are the fundamental mechanism needed to extend the behavior of user-defined types. Supporting the evolution of type behavior is one of the most important requirements of a persistent type system.

Sourced functions. The simplest way for a user to define a new function is to declare the new function to be *sourced* on an existing function, a feature that is not supported by SQL3 or by any relational DBMS that provides user-defined functions. This is analogous to defining a distinct type based on an existing type. When sourced, the new function must have the same number of arguments as its source function, each of its argument types must be capable of being cast to the corresponding argument type of the source function, and its return type must be capable of being cast from the return type of the source function.

Sourced functions provide an easy way to create user-defined functions for a distinct type from existing functions of the base type. Since operators such as + and − are treated as functions, sourced functions can also be used to define such operators for a distinct type and thereby make use of the efficient implementation of operators for predefined types. For example, suppose that money is a distinct type based on the built-in type decimal (8,2). The following statement could be used to make the + operator of the decimal type applicable to the money type:

```
CREATE FUNCTION "+" (money, money)
RETURNS money
SOURCE "+" (decimal, decimal);
```

Based on this definition, the expression salary + bonus, where salary and bonus are columns of type money, would be interpreted as equivalent to the expression money(decimal (salary) + decimal (bonus)). (Note that the infix nature of + is implicitly inherited by the sourced function, i.e., + on money.) If the user defines no function "*" (money, money), the expression salary * bonus will be treated as an error. In this way, a distinct type can selectively inherit the semantics of its source type that make sense for the distinct type.

Sourced, SQL, and external functions can support user-defined functions.

SQL functions. In order to create a new function that is not based on an existing function, it is necessary to write a body that specifies the semantics of the function. Such function bodies can be written either in SQL itself (an SQL function) or in a host programming language (an external function). Writing the body of a function in SQL has the advantages that the function can be executed entirely within the database context, avoiding the overhead of switching to a host-language context, and that user-defined types (especially ADTs) can be passed as arguments to the function without having to decompose them into more primitive types understood by the host programming language. Moreover, functions written in SQL can be optimized by the DBMS optimizer.

Some commercial database products permit users to invoke stored procedures written in a language that is native to the database product. The draft SQL3 standard includes several new procedural features in SQL, such as assignment statements, IF statements, LOOP statements, CASE statements, and BEGIN-END blocks, that may help minimize the proliferation of incompatible procedural languages. The incorporation of these constructs into SQL3 will enable users to write portable user-defined functions that can be stored and executed in any relational DBMS conforming to the SQL standard. For example, suppose that an ADT has been defined to represent a point using rectangular coordinates:

```
CREATE TYPE point
    (x FLOAT,
     y FLOAT);
```

A user could simulate polar coordinates by writing the following two SQL3 functions that operate on instances of the point type. Note that these functions depend on the existence of other functions named SQRT and ARCTAN (which might be either built-in or user-defined functions).

```
FUNCTION RHO (:p POINT) RETURNS FLOAT
    BEGIN
        DECLARE :temp FLOAT;
        IF :p IS NULL
          THEN SET :temp = NULL
          ELSE SET :temp = SQRT( (X(:p) * X(:p))
                                 + (Y(:p) * Y(:p)) );
        RETURN :temp;
    END;

FUNCTION THETA (:p POINT) RETURNS FLOAT
    BEGIN
        IF  :p IS NULL
          THEN RETURN NULL
          ELSE RETURN ARCTAN( Y(:p) / X(:p) );
    END;
```

External functions. A user-defined function whose body is written in a host programming language is called an *external function.* Since language types are not supported in the current draft of SQL3, it is necessary to decompose an ADT instance into its primitive components before passing it to an external function. If language types, as described earlier in this paper, are part of a type subsystem, ADTs can be passed to external functions in the form of language-type instances with a representation that is appropriate for the host programming language in which the body of the external function is written. Thus, the support of language types would enable the use of external functions to manipulate ADTs without the need to decompose them, thereby allowing ADTs to maintain their behavior even after "crossing the border" between the database system and the host programming language.[2]

The following example illustrates how the RHO and THETA functions previously defined might be implemented as external functions without language types by decomposing their parameters into primitive types known by the host language. The two functions are written in C. The function parameters correspond to the x- and y-components of the input type point, followed by a null indicator for the input point, followed by pointers to output buffers for the function result and its null indicator.

```
void get_rho (float x, float y, int nullIn,
              float *rho, short *nullOut)
  {
  if  (nullIn < 0) *nullOut = -1;
  else
    {
    *rho = sqrt((x * x) + (y * y));
    *nullOut = 0;
    }
  }

void get_theta (float x, float y, int nullIn,
                float *theta, short *nullOut)
  {
  if (nullIn < 0) *nullOut = -1;
  else
    {
    *theta = atan(y/x);
    *nullOut = 0;
    }
  }
```

Once these C function bodies have been written, the following SQL statements can be executed to register the database functions RHO and THETA and to instruct the database system that they are implemented by external functions written in C. The database system will search for these functions in a designated library, and will call them with a parameter decomposition convention as in the example above.

```
CREATE FUNCTION rho(POINT)
    RETURNS FLOAT
    EXTERNAL NAME get_rho
    LANGUAGE C;

CREATE FUNCTION theta(POINT)
    RETURNS FLOAT
    EXTERNAL NAME get_theta
    LANGUAGE C;
```

Dynamic binding. In traditional relational database systems, each function has a unique name. For example, the SUBSTR function in the IBM DATABASE 2* (DB2*) system is a built-in function

that takes a character string and two integers as parameters, and returns a character string. However, in most modern object-oriented languages, a user can *overload* a function name by defining several functions with the same name but with different parameter types. Overloading is an important feature for providing type-specific behavior within a subtype family (i.e., the set of all types that are subtypes of a given supertype). For example, suppose that t_bus_address is a subtype of t_address, and that the following functions have been defined:

```
assess_land_value (t_address)
    returns decimal (10,2)
assess_land_value (t_bus_address)
    returns decimal (10,2)
```

The assess_land_value function for business addresses may be a more specialized function that takes into account the appropriateness of the property for commercial buildings.

For any given function invocation and a set of overloaded functions, the function is selected whose parameter types best match the types of the actual arguments of the invocation. Because of subtypes and substitutability, the process of selecting the best function for a given invocation cannot always be completed at compile time, since a run-time argument may be an instance of a subtype of the corresponding formal parameter for the function. Consequently, to guarantee the invocation of the function that is the best match for a set of arguments, function selection must be done at run time. Thus, if an instance of a subtype is passed at run time to a function, the function that is defined on the subtype is selected for execution rather than the function defined on the static type of the argument. However, the function invocation is still completely type checked at compile time to make sure that no type errors will occur at run time independent of which function is selected for execution. For example, suppose a table named customers has a column named location, of type t_address. The following query invokes a function on the values in the location column:

```
SELECT assess_land_value(location)
FROM   customers
WHERE  balance_owed > 10000;
```

Since the value of location in a given row of the customers table might be either a t_address or a t_bus_address, it is not possible to select between the two instances of the assess_land_value function at compile time. In such a case, the final selection of the function to be invoked is deferred until run time and is based on the types of the actual function arguments. This process is called *dynamic binding*.

The draft SQL3 performs dynamic binding of functions based on types of all their arguments.

Overloading and dynamic binding of functions are supported by most object-oriented languages. However, in many of these languages (e.g., C++,[9] Smalltalk,[10] and Eiffel[11]), dynamic binding is based on the type of only a single function argument. In C++, for example, the expression x→foo(y,z) invokes the function foo on object x with arguments y and z. In this example, run-time function selection is based only on the type of the (implicit) argument x; arguments y and z are passed to the function but do not participate in selection of the function instance. Consequently, in these languages, functions like

```
draw (window, polygon) returns float
draw (window, triangle) returns float
draw (square-window, polygon) returns float
draw (square-window, triangle) returns float
```

cannot be dynamically selected based on the actual types of both window and polygon.

The draft SQL3 language is more flexible than the above languages in that it performs function selection based on the dynamic types of all the arguments of a function. An invocation draw(:x, :y) in which the declared type of x is window and the declared type of y is polygon might result in selection of any of the above four functions, based on the actual (run-time) type of both the x and y

arguments. Algorithms for dynamic binding of functions based on the types of all their arguments (as in the Common Lisp Object System, CLOS)[12] have been developed in experimental IBM extended relational systems[2,13] and recently accepted for incorporation into SQL3.[14]

The draft SQL3 includes a trigger capability.

Managing database rules

As database systems become more advanced, they are used as repositories not only for data, but also for the rules (constraints, assertions, and triggers) that are associated with data. Rules were first introduced into the standard SQL language by the Integrity Feature of SQL89,[15] which allowed users to express *check* constraints to ensure the validity of data on entry to the database, and *referential integrity* constraints to guarantee that all the values of a foreign key are matched by primary-key values in another table. With the Integrity Feature of SQL89, any attempt to violate a check constraint or referential integrity constraint causes the attempted database update to be refused. The SQL92 Standard[16] extends the rules capability of the SQL language in several important ways. It allows a constraint to be defined for a *domain*, which in turn can be used in the definitions of many database columns. It also provides a richer set of actions for referential integrity constraints. For example, if a primary key is updated, the update can be automatically cascaded to all the matching foreign keys in dependent tables.

A number of researchers[17–23] have experimented with *active databases*, in which an arbitrary sequence of actions can be *triggered* by the detection of some condition in the database. Triggers are also becoming available in relational database products such as Sybase, ORACLE, and INGRES. In general, a trigger consists of a *triggering action* (e.g., insertion, deletion, or update), a *condition* that must be satisfied in order for the trigger to become effective (generally, an SQL predicate), and an *action* (generally, any sequence of SQL statements). The testing of the trigger condition and the invocation of its action may be immediate or deferred (delayed until the end of the current transaction). A deferred evaluation is important when the user wants the condition or action of the trigger to be executed against changes in the database that have been caused by several SQL statements (and not only by the triggering action). The trigger condition and action statements need access to both the old (pre-update) and new (post-update) values of the database entries whose modification caused the trigger to be invoked. Triggers can be used to monitor important conditions in the database and can also be used to generate an audit trail of database updates.

Assertions are another important part of a database rule subsystem. They express conditions that need to be satisfied by the database at all times. Because they do not have an action part, they (in contrast to triggers) cannot be used to perform corrections that might be necessary to maintain the database in a consistent state. Assertions that apply to the database as a whole (e.g., that the average salary of employees must not exceed a given value), are particularly easy to express in a set-oriented relational language such as SQL, and are well-suited for implementation by the access path optimizers that are a standard part of all relational systems.

Assertions and triggers increase the value of stored data by guaranteeing integrity and increasing semantic content. Relational database systems are well-suited for the specification of assertions and triggers in an easy-to-understand, declarative syntax. By associating triggers and assertions with the data themselves, extended relational systems make it unnecessary to repeat the logic of the triggers and assertions in every program that manipulates the data, thus protecting database integrity and making it easier to develop correct applications.

A version of an active database (trigger) capability is included as part of the draft SQL3 standard. The example below illustrates the definition of several constraints and a trigger, using the syntax under consideration for SQL3. These statements guarantee that all salaries and commissions are positive numbers, that gender data consist only of certain codes, that employee numbers are unique,

and that all employees are assigned to a valid department. It also causes all employees who earn a commission equal to or greater than 20 percent of their salaries to be entered into a separate "winners" table.

```
CREATE DOMAIN money
    DECIMAL(8,2)
    CHECK (value >= 0);

CREATE TABLE employee
     (name VARCHAR(28),
      empno DECIMAL(6,0) PRIMARY KEY,
      deptno CHAR(4) REFERENCES dept(deptno),
      sex CHAR(1) CHECK (sex IN ('F', 'M')),
      salary MONEY,
      commission MONEY);

CREATE TRIGGER salestrig1
    AFTER UPDATE OF commission
    ON employee
    WHEN (commission > .2 * salary)
         INSERT INTO winners
         VALUES (name, deptno, CURRENT DATE)
```

Performance challenges

Performance is a critical concern for database systems supporting advanced applications such as computer-aided software engineering (CASE), computer-aided design / computer-aided manufacturing (CAD/CAM), engineering and scientific applications, office automation, and hypermedia. Applications such as these tend to have the following characteristics:

- A high degree of interactivity. For example, the generation of a CAD display may require processing of thousands of objects within a human interaction time.[24]
- A complex structure where data may include many cyclic or recursive relationships among objects. Traversal of these relationships may require processing of hundreds or thousands of objects per second.
- A need for navigational data access. Applications often need to traverse graphs of related objects, as when rearranging connected components on a circuit board. During this process, objects may be visited several times to perform complex operations. It is sometimes awkward to express this kind of navigational access in terms of value-based relational join operations.
- A large amount of data. The sheer bulk of multimedia data leads to rigorous performance requirements, particularly for real-time applications such as video display.

A consortium named the Transaction Processing Performance Council has developed several well-known database performing benchmarks for online transaction processing (TPC-A, TPC-B, and TPC-C) and for decision support applications (TPC-D).[25] Recently, benchmarks have been developed to measure the performance of other kinds of applications:

- The Cattell benchmark[26] measures database performance on engineering and computer-aided design (CAD) applications. The test database of the Cattell benchmark consists of a set of parts in a bill-of-materials application. Each part is connected to three other randomly selected parts. Measurements are made on random lookup, traversal of all connected parts, and insertion of parts.
- The 007 benchmark[27] is designed to provide a comprehensive profile of the performance of an object-oriented database system. The test database is built on a set of composite parts corresponding to a VLSI (very large-scale integrated) CAD application. The benchmark consists of a combination of pointer traversals, different kinds of updates (update to indexed and nonindexed objects, repeated updates, sparse updates, updates of cached data, and the creation and deletion of objects), and different types of queries (exact match, aggregation, etc.)
- The Sequoia 2000 benchmark[28] is aimed at geographic information system (GIS) applications. The test database consists of raster data, point data, polygon data, and directed graph data. The benchmark consists of data load, raster queries, point and polygon queries, spatial joins and recursion. Measurements have been made with POSTGRES (an extended relational DBMS prototype), GRASS (a public domain geographic information system), and IPW (a raster-oriented image processing package).

The next two sections discuss techniques that can be used with extended relational systems to address the performance requirements of advanced applications. Some of the extensions have been prototyped and measured. From the introduction of DB2 Version 1.1 in 1985 until DB2 Version 2.3 in 1991, we have seen a performance improvement

of 36 times in transaction throughput for the DB2 benchmark.[29] Faster processors and larger quantities of memory have contributed a factor of 14–18 to this improvement, and software enhancements contributed further for a factor of 2–3. We expect that hardware speed and capacity will continue to improve dramatically, especially as database systems take advantage of parallel architectures. For example, DB2 Version 3.1 can exploit up to eight processors to execute queries on an IBM ES9000* Model 982.

Language extensions. Language extensions not only enhance the expressive power of the SQL language, but also provide information that helps the database system to optimize query response time. We will illustrate this point using the recursive query syntax under consideration for SQL3. Suppose that a MATERIALS table has columns PART, SUBPART, and QUANTITY. A large assembly such as an airplane wing might contain many subassemblies such as ailerons and landing gear, which in turn might contain common parts such as rivets and hinges. The following recursive query finds the total quantity of each part that is used in assembling a wing, summarizing all levels of assembly. The query works by computing a temporary table that includes all the first-level subassemblies of a wing, and then joining the temporary table to the MATERIALS table repeatedly until all the lower-level subassemblies have been considered.

```
SELECT part, subpart, sum(quantity)
FROM
    (SELECT  part, subpart, quantity
     FROM    materials
     WHERE  part = 'wing'

     RECURSIVE UNION
             temp (part, subpart, quantity)

     SELECT  t.part, f.subpart, t.quantity * f.quantity
     FROM    temp t, materials f
     WHERE  t.subpart = f.part)
GROUP BY part, subpart;
```

Recursive queries have been implemented in the Starburst system by means of an automatic query rewrite algorithm that transforms the recursive query into another form, eliminating redundant computation of subparts. Measurements have shown the resulting query executes 300 times faster than the original nonoptimized recursive query in some cases.[30]

Other language extensions described in this paper also have positive implications for query performance. For example, an extensible type system allows specialized methods to be incorporated into query predicates, thus increasing the semantic content of stored data and reducing the number of times the interface between the host language and the database system must be crossed in processing a query.

Similarly, a rules subsystem moves the responsibility for protecting database integrity from application programs into the database itself. In addition to improving the level of protection, this approach results in improved performance for two reasons: first, the number of interactions between database and application programs is reduced; and second, the database system has an opportunity to group and optimize the checking of multiple rules.

Improved data access facilities. There are many ways to improve the efficiency of accessing data. For example, it is important that data items be stored near each other physically if they are frequently used together. As a result, fewer logical blocks of data, or pages, need to be moved and buffered when these items are transferred to or from the physical storage medium such as a disk. Physical clustering within a table has been an important feature of relational systems for many years.[31] More recently, the Starburst relational system has demonstrated significant performance gains by using cross-table clustering between related tuples.[32]

Relational systems retrieve related objects via join operations. The repertoire of join evaluation methods has been improved continuously[33–36] and can be further extended to include join indexes and links. A join index spans two tables, and each of its entries represents one row of an equijoin between the tables. (Equijoin occurs when the comparison operator is equality.) Join indexes allow faster read access at the expense of additional costs for index maintenance during update and insert operations. Links are logical or physical pointers from a row to related rows. The Starburst relational system has recently demonstrated significant improvements in join performance through the use of links.[29]

The number of I/O operations required to process a given query can also be reduced by processing all the common subexpressions in a single pass over the data. For example, experiments have shown that the processing time of the following query can be cut in half by common-subexpression optimization. The query, which summarizes a set of high and low-priority orders, is a part of the TPC-D benchmark under consideration by the Transaction Processing Performance Council.

```
SELECT  l_shipmode, count(*), 'high priority'
FROM    orders, lineitem
WHERE   o_orderkey = l_orderkey AND
        (o_orderpriority = '1-urgent' OR
                o_orderpriority = '2-high') AND
        (l_shipmode = 'mail' OR
                l_shipmode = 'ship') AND
        l_commitdate < l_receiptdate AND
        l_shipdate < l_commitdate AND
        l_receiptdate BETWEEN 94001 AND 94365
GROUP BY l_shipmode

UNION  ALL

SELECT  l_shipmode, count(*), 'low priority'
FROM    orders, lineitem
WHERE   o_orderkey = l_orderkey AND
        o_orderpriority <> '1-urgent' AND
        o_orderpriority <> '2-high' AND
        (l_shipmode = 'mail' OR
                l_shipmode = 'ship') AND
        l_commitdate < l_receiptdate AND
        l_shipdate < l_commitdate AND
        l_receiptdate BETWEEN 94001 AND 94365
GROUP BY l_shipmode

ORDER BY 2 DESC;
```

Several performance optimizations are possible in the processing of large objects, including the following:

- Direct transfer of data from disk to application, bypassing DBMS buffers
- Bypass logging (optional), thus avoiding excessive I/O traffic on the log file
- Allow users to define their own functions on long fields and to use them in queries. This enables testing of search predicates on long fields to be moved from application programs into the database engine, thus eliminating unnecessary transfers of long fields between the database and the application.
- Defer materialization of long fields as long as possible, using the handle concept as described in a previous section of this paper.

Conclusions

Relational database systems have become dominant in the industry because they offer the advantages of data independence, high-level set-oriented query languages, automatic optimization, multiple views of shared data, and support for a variety of host programming languages. These advantages are as important today as ever. However, a new generation of applications is appearing that will stress today's database systems in unprecedented ways. This paper has discussed several ways in which relational database technology is evolving to meet the challenges of this new generation of applications. We have described how relational database systems can be extended to support user-defined types, functions, and rules, and we have discussed several different approaches for improving the performance of extended relational systems. The advantages of extended relational DBMSs can be summarized as follows:

- Upward-compatible type extensions to standard relational systems:

  Because user-defined types and functions are used in exactly the same way as system-provided types and functions, they are an upward-compatible extension to today's relational systems. This approach minimizes the necessary extensions to standard SQL and enriches the behavior of the data elements over which the relational database is defined. User-defined types and functions are also an important part of the object-oriented extensions in the draft SQL3 standard.

- Multifunctions:

  In SQL3, the selection of a function to be invoked is based on the dynamic types of all its arguments. This is in contrast to languages such as C++,[9] Smalltalk,[10] and Eiffel,[11] which perform dynamic function selection based on the type of a single argument. Algorithms have been developed[11] whereby the type-safety of a multifunction can be guaranteed at compile time even though the actual function selection is performed at run time.

- Preservation of database type behavior in the host language:

  By using language types, instances of database types can be fetched into application programs without losing their type-specific behavior. Moreover, instances of database types can be converted into representations that are appropriate to a host language. This allows existing libraries written in host programming languages to be used to manipulate instances of database types in the host language.

- Integration with multiple host languages:

  The type extensions described in this paper are accessible from multiple host languages, in keeping with the usual practice of relational systems. Type libraries that are written in a host language can be used against data retrieved from the database, and functions defined in the database can be applied to data created in an application program. Thus the semantics of a user-defined type can be shared across multiple programming languages, using the database as a common repository.

- Declarative rules:

  The declarative approach in which the system automatically enforces constraints and rules is a distinctive feature of extended relational systems.[37]

- Improved support for very large objects:

  The use of handles to represent BLOB values improves both the performance and the convenience of BLOBs by deferring their materialization for as long as possible.

Many of the techniques described in this paper have been demonstrated by experimental prototypes at the IBM Almaden Research Center[2,7,38] and at the IBM Database Technology Institute. By means of these techniques, we believe that relational systems can add value to stored data and continue to deliver superior function in a changing application environment.

Acknowledgments

The ideas presented in this paper are the work of many people at several IBM locations. We acknowledge the contributions of the members of the IBM Database Technology Institute, the Starburst and Polyglot research projects at the IBM Almaden Research Center, the development staff at the IBM Almaden, Santa Teresa, and Toronto laboratories, and the IBM SQL Language Council.

Cited references

1. *Database Language SQL3 (Working Draft)*, Jim Melton, Editor, ANSI Database Committee (X3H2), American National Standards Institute, New York (September 1993).
2. L. DeMichiel, D. Chamberlin, B. Lindsay, R. Agrawal, and M. Arya, "Polyglot: Extensions to Relational Databases for Sharable Types and Functions in a Multi-Language Environment," *Proceedings of the Ninth International Conference on Data Engineering,* Vienna; IEEE Computer Society Press, Los Alamitos, CA (1993), pp. 651–661. Also available as Research Report RJ-8888, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1992).
3. G. Gardarin, J-P. Cheiney, G. Kiernan, D. Pastre, and H. Stora, "Managing Complex Objects in an Extensible Relational DBMS," *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam; Morgan Kaufmann Publishers, Incorporated, Palo Alto, CA (August 1989), pp. 55–66.
4. V. Linnemann, K. Kuespert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Suedkamp, G. Walch, and M. Wallrath, "Design and Implementation of an Extensible Database Management System: Supporting User Defined Data Types and Functions," *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Los Angeles, CA; Morgan Kaufmann Publishers, Incorporated, Palo Alto, CA (August 1988), pp. 294–305.
5. F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, IEEE Computer Society, Washington, DC (September 1986).
6. M. Stonebraker, "Inclusion of New Types in Relational Database Systems," *Proceedings of the Second International Conference on Data Engineering*, Los Angeles, CA; IEEE Computer Society, Washington, DC (1986).
7. T. J. Lehman and B. G. Lindsay, "The Starburst Long Field Manager," *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam; Morgan Kaufmann Publishers, Incorporated, Palo Alto, CA (August 1989), pp. 375–383.
8. P. Cotton, T. Lehman, N. Mattos, and F. Pellow, *Large Object Strings*, ANSI X3H2-93-341, Rev. 1, ANSI Database Committee (X3H2), American National Standards Institute, New York (July 1993).
9. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Co., Reading, MA (1987).
10. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Co., Reading, MA (1983).
11. B. Meyer, *Eiffel: The Language*, Technical Report TR-

EI-17/RM, Interactive Software Engineering Inc., 270 Storke Rd., Suite 7, Goleta, CA 93117 (August 1989).
12. D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczeles, and D. A. Moon, *Common LISP Object System Specification*, ANSI X3J13 Document 80-002R; also published in *ACM SIGPLAN Notices* **23,** special issue (September 1988).
13. R. Agrawal, L. DeMichiel, and B. Lindsay, "Static Type Checking of Multi-Methods," *OOPSLA '91 Proceedings*, Phoenix, AZ; Association for Computing Machinery, New York, pp. 113–128.
14. L. DeMichiel and N. Mattos, *Enhancing the Algorithm for Subject Routine Determination to Better Support Multi-Functions and Multiple Inheritance*, ANSI X3H2-93-373, Rev. 1, ANSI Database Committee (X3H2), American National Standards Institute, New York (September 1993).
15. *Database Language SQL with Integrity Enhancement*, ANSI Standard X3.135-1989, American National Standards Institute, New York.
16. *Database Language SQL*, ANSI Standard X3.135-1992, American National Standards Institute, New York.
17. D. Chamberlin and K. Eswaran, "Functional Specifications of a Subsystem for Database Integrity," *Proceedings of the First International Conference on Very Large Data Bases*, Framingham, MA; Association for Computing Machinery, New York (September 1975).
18. M. Hsu, R. Ladin, and D. McCarthy, "An Execution Model for Active Data Base Management Systems," *Proceedings of the Third International Conference on Data and Knowledge Bases, Improving Usability and Responsiveness*, Jerusalem; Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (June 1988).
19. M. Stonebraker, E. Hanson, and S. Potamiano, "The POSTGRES Rule Manager," *IEEE Transactions on Software Engineering* **14**, 7, 897–907 (July 1988).
20. U. Dayal, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M. Livny, R. Jauhari, M. Carey, B. Blaustein, A. Buchmann, and U. Chakravarthy, *HiPAC: A Research Project in Active, Time-Constrained Database Management—Final Technical Report*, Technical Report XAIT-89-02, Xerox Advanced Information Technology (July 1989).
21. M. Stonebraker, A. Jhingran, J. Goh, and S. Potaminanos, "On Rules, Procedures, Caching and Views in Data Base Systems," *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Nashville, TN; Association for Computing Machinery, New York (1990), pp. 281–290.
22. U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona; Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (September 1991), pp. 469–478.
23. J. Widom, R. J. Cochrane, and B. G. Lindsay, "Implementing Set-Oriented Production Rules as an Extension to Starburst," *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona; Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (September 1991), pp. 275–286.
24. W. Harrison and H. Ossher, *A Comparison and Evaluation of Five Persistent Object Stores: Versant OODBMS™, ObjectStore™, GemStone™, CLORIS, and RPDE/OS*, Research Report RC-16724, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (April 1991).
25. *The Benchmark Handbook for Database and Transaction Processing Systems*, Jim Gray, Editor, Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (1991).
26. R. Cattell, "An Engineering Database Benchmark," *The Benchmark Handbook for Database and Transaction Processing Systems*, Jim Gray, Editor, Morgan Kaufmann Publishers, Incorporated, San Mateo, CA (1991).
27. M. Carey, D. DeWitt, and J. Naughton, "The 007 Benchmark," *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, Washington, DC; Association for Computing Machinery, New York (June 1993), pp. 12–21.
28. M. Stonebraker, J. Frew, K. Gardels, and J. Meredith, "The Sequoia 2000 Storage Benchmark," *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, Washington, DC; Association for Computing Machinery, New York (June 1993), pp. 2–11.
29. D. Hauser and A. Shibamiya, "Evolution of DB2 Performance," *InfoDB* **6,** No. 4, 2–13 (Summer 1992). Published by Database Associates International, P.O. Box 215, Morgan Hill, CA 95038.
30. H. Pirahesh, G. Kiernan, R. Agrawal, B. Lindsay, G. Lohman, and J. McPherson, *Implementing Recursive Query Processing in a Relational DBMS*, unpublished manuscript, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120.
31. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference*, Boston, MA; Association for Computing Machinery, New York (June 1979), pp. 23–34.
32. M. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson, "An Incremental Join Attachment for Starburst," *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia (September 1990), pp. 662–673. Also Research Report RJ-7544, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120.
33. K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations," *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore (August 1984), pp. 323–333. Department of Computer Science, Norwegian Institute of Technology, University of Trondheim, N-7074 Norway.
34. D. J. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Eleventh International Conference on Very Large Data Bases*, Stockholm (1985), pp. 151–164. Department of Computer Science, University of Wisconsin, Madison, WI 53706.
35. P. Valduriez, "Join Indices," *ACM Transactions on Database Systems* **12**, No. 2, 218–246 (June 1987).
36. J. Cheng, D. Haderle, R. Hedges, B. Iyer, T. Messinger, C. Mohan, and Y. Wang, "An Efficient Hybrid Join Algorithm: A DB2 Prototype," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan; IEEE Computer Society Press, Los Alamitos, CA (April 1991).
37. The Committee for Advanced DBMS Function, "Third-Generation Data Base System Manifesto," *SIGMOD Record* **19**, 3 (September 1990).
38. G. Lohman, B. Lindsay, H. Pirahesh, and B. Schiefer, "Extensions to Starburst: Objects, Types, Functions, and

Rules," *Communications of the ACM* **34**, No. 10, 95–109 (October 1991).

Accepted for publication February 14, 1994.

Josephine M. Cheng *IBM Software Solutions Division, Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: chengjm@vnet.ibm.com).* Ms. Cheng is a Senior Technical Staff Member and a manager at the Database Technology Institute at the IBM Santa Teresa Laboratory, responsible for advanced technology in IBM database products. Her interests include object-oriented technology, multimedia technology, and new application areas. Previously, Ms. Cheng participated in development of the DB2 relational database system as a designer, implementer, and manager. She has filed five patents in query processing and optimization. Ms. Cheng received the B.S. degree in mathematics and computer science and the M.S. degree in computer science from the University of California, Los Angeles, in 1975 and 1977 respectively.

Nelson M. Mattos *IBM Software Solutions Division, Santa Teresa Laboratory, 555 Bailey Ave, San Jose, California 95141 (electronic mail: mattos@vnet.ibm.com).* Dr. Mattos is a database language architect at the Database Technology Institute at the IBM Santa Teresa Laboratory, working on extended relational database systems. He is also IBM's SQL Standard Project Authority and a member of the ANSI Technical Committee X3H2 for Database, as well as a U.S. representative to the International Organization for Standardization (ISO) committee for databases. He has contributed extensively to the design of SQL3. Prior to joining IBM, Dr. Mattos worked for several years as the leader of a large project on object-oriented and knowledge base management systems at the University of Kaiserslautern in Germany. He received a Ph.D. in computer science from the University of Kaiserslautern in 1989. He has published several papers and a book on data and knowledge management.

Donald D. Chamberlin *IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: chamberlin@almaden.ibm.com).* Dr. Chamberlin is a research staff member at the IBM Almaden Research Center and an adjunct professor of computer engineering at Santa Clara University. His current work is focused on object-oriented extensions to relational database systems. He is an ACM Fellow and was one of the original developers of the SQL data language. He received a Ph.D. in electrical engineering from Stanford University and joined IBM at the T. J. Watson Research Center in Yorktown Heights, New York. He has published numerous papers on database management and document processing, and has received the ACM Software System Award and two IBM Corporate Awards for his contributions to relational database systems.

Linda G. DeMichiel *IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: lgd@almaden.ibm.com).* Dr. DeMichiel is a research staff member at the IBM Almaden Research Center, where her current work is focused on the design of object-oriented type systems for databases. She is also a contributor to the ANSI X3H2 Technical Committee for Databases. Prior to joining IBM, she worked at Lucid Inc., on the design and specification of the Common Lisp Object System, and at Xerox Corp., on the Pilot operating system. She received an A.B. degree from Oberlin College and a Ph.D. degree in computer science from Stanford University.

Reprint Order No. G321-5542.