O-O, What Have They Done to DB2?

Michael Carey
Don Chamberlin
Srinivasa Narayanan
Bennet Vance
IBM Almaden Research Center

Doug Doole
Serge Rielau
Richard Swagerman
IBM Toronto Laboratory

Nelson Mattos
IBM Database Technology Institute
Santa Teresa Laboratory

Abstract

In this paper, we describe our recent experiences in adding a number of object-relational extensions to the DB2 Universal Database (UDB) system as part of a research and development project at the IBM Almaden Research Center. In particular, we have enhanced DB2 UDB with support for structured types and tables of these types, type and table hierarchies, references, path expressions, and object views. In doing so, we have taken care to design and implement the extensions in such a way as to retain DB2's ability to fully optimize queries and (in our next step) to support business rules and procedures through the provision of constraints and triggers. We describe each of the SQL language extensions that we have made, discuss the key performance tradeoffs related to the design and implementation of these features, and explain the approach that we ended up choosing (and why). Most of the features described here are currently shipping as part of Version 5.2 of the DB2 UDB product. We end this paper with a summary of the current status of our work and a discussion of what we plan to tackle next.

1 Introduction

The introduction of the relational model [6] revolutionized the information systems world by providing a simple, high-level data model and the foundation for declarative query interfaces. Relational database systems, with their separation of the logical schema (tables) from the underlying physical schema (storage and index structures), together with their support for alternative views of a given logical schema, have been very successful in providing a high level of data independence that has led to significant productivity gains for both application programmers and end users. The past 15–20 years of research in the database area, initiated by the relational revolution, have brought us to an era where most modern relational database systems offer efficient query optimization and execution strategies, excellent levels of multi-user performance and robustness through well-tuned buffer and transaction management subsystems, view facilities for alternative conceptual schemas and flexible authorization, and native business logic support through the provision of declarative constraints, triggers, and stored procedures [12]. Finally, the declarative nature and set-orientation of relational query languages laid a natural foundation for research on parallelization of database operations; as a result, parallel relational database systems have become what is by far the most significant commercial success story in the area of parallel computing [7].

Despite this success story, the world has continued to place ever-increasing demands on database technology. One reason for this is the appearance of interesting new data types (text, images, audio, video, spatial data) and applications wishing to use database systems to manage them in large quantities. A second reason is the mismatch between the complexity of modern enterprises and the spartan simplicity of the relational model: enterprises have entities and relationships (versus tables), variations within a given kind of entity (versus the homogeneity of relational tables), and both single- and multi-valued attributes (versus relational normalization rules). A third reason is commercial growth in applications that wish to use database systems to manage large quantities of highly complex interrelated data objects, including

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

CAD/CAM systems, web servers, and digital libraries, to name a few. This has led the research community to look for new solutions, particularly through "objects," for the past decade or so [1]. In particular, this growth has led relational database system researchers and vendors to look at the option of adding object-oriented extensions to the relational model and its query languages. As a result, relational database systems are evolving into *object-relational* database systems that provide such features as an extensible type system, inheritance, support for complex objects, and rules [13].

IBM's DB2 Universal Database system, IBM's version of DB2 for Unix, NT, Windows, and OS/2 platforms (both serial and parallel), has been making the transition into an object-relational database system since the debut of DB2 Version 2 for Common Servers in 1995. DB2 V2 incorporated various new technologies developed in the context of the Starburst research project at IBM Almaden [8]. In terms of object-relational extensions, DB2 V2 included significant new features in the areas of user-defined column types (UDTs, also referred to as "distinct types"), user-defined functions (UDFs), and triggers. The DB2 Universal Database (UDB) System, which became available as DB2 Version 5 in late 1997, added new support for utilizing these features on parallel platforms by merging DB2 V2 with DB2 Parallel Edition, a previously separate product for MPP hardware platforms. In addition, DB2 UDB includes a set of "extenders" for dealing with commonly interesting new data types including text, image, and audio; these extenders currently use a mix of UDTs, UDFs, and triggers to provide their functionality.

For about two years now, the OSF ("Object Strike Force") project, a joint effort between the IBM Almaden Research Center and the IBM Database Technology Institute, has been working to add another dimension of object-relational functionality to DB2 UDB. In particular, we have been extending UDB with support for user-defined structured types with inheritance, tables and subtables of these types, object ids and references, path expressions, and object views. This support made its public debut in DB2 UDB Version 5.2 in September of 1998, and more is coming. In adding these features to UDB, we have taken care to ensure that our extensions provide a step forward in UDB's data modeling and data manipulation functionality without dictating a corresponding step backward in terms of its performance or the provision of advanced features such as automatic query optimization, constraints, or triggers [14, 10]. Our end goal is to evolve UDB into a strong platform for general-purpose complex object management. In this paper, we share some of the experiences that we have had so far in the process. We describe the SQL extensions that we have made, discuss some performance tradeoffs that we have faced in the design and implementation of these extensions, and discuss particular choices that we made (and why) in adding these features to UDB at Almaden. Our hope is that this paper will be of interest to database students and practitioners; it should also be of interest to researchers interested in monitoring commercial progress in the area of object-relational databases. In addition, we have been heavily involved in reshaping the SQL99 (known as SQL3 until recently) standard over the past 1–2 years, and the bulk of our DB2 extensions are SQL99-compliant; thus, this paper also provides a look at the object model and query facilities in SQL99 as it stands today.

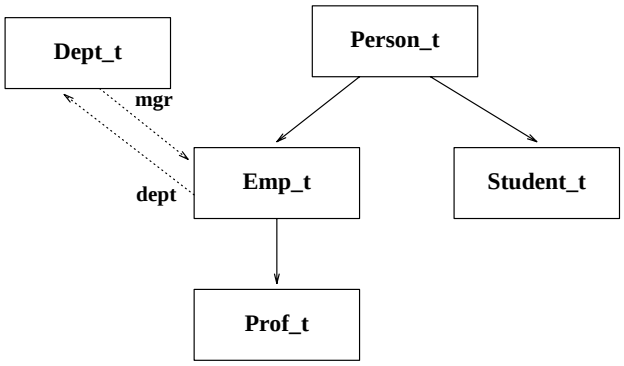
The remainder of this paper is organized as follows: In the next few sections, we discuss our SQL language extensions; we start with our basic DDL extensions, turn to the associated DML extensions, and then turn to advanced features such as object views, constraints, and triggers. Throughout, we discuss the design considerations that led us to make the choices that we made. Following the language sections, the next section of the paper discusses how we approached the implementation of certain key object features. Again, we attempt to share some of the reasoning that led to the choices that we ended up making. Finally, the last two sections of the paper discuss where we are with UDB today, roughly how the resulting system compares to other vendors' offerings, and lists some new features that we are either currently exploring or planning to explore.

2 Basic SQL Data Definition Language Extensions

We have extended SQL's data definition language (DDL) into object-oriented (O-O) territory by adding a number of features commonly found in O-O database systems. In this section we describe our extensions by example, using a very simple university database.

One of the fundamental features that we have added to UDB is a facility that allows users to define structured types via a new **create type** statement. Two basic entity types in a university schema are people (Person_t) and departments (Dept_t). University people come in various flavors, such as university employees (Emp_t) and students (Student_t). Within employees, there may again be various flavors; let's suppose that there are just regular employees and professors (Prof_t). Let's assume that each employee works in a department, and that each department is managed by an employee. (We ignore the many other relationships that might exist among these types to keep the example simple.) Figure 1(a) shows the resulting entity and relationship types graphically.

Our O-O extensions to SQL's DDL enable the user to directly translate the entities and relationships above into a set of structured types and subtypes with references. In UDB, these type definitions would be



(a) Schematic of university entities and relationships

```
create type Person_t as (
    name Varchar(40),  birthyear Integer
) mode db2sql;

create type Emp_t under Person_t as (
    salary Integer
) mode db2sql;

create type Prof_t under Emp_t as (
    rank Varchar(10),  specialty Varchar(20)
) mode db2sql;

create type Student_t under Person_t as (
    major Varchar(20),  gpa Decimal(5,2)
) mode db2sql;

create type Dept_t as (
    name Varchar(20),  budget Integer,
    headcnt Integer,  mgr Ref(Emp_t)
) mode db2sql;

alter type Emp_t add attribute dept Ref(Dept_t);
```

(b) DDL statements to create the university types

Figure 1: A structured-type hierarchy

specified (filling in their attribute details) as shown in Figure 1(b). The first statement creates the type Person_t with attributes name and birthyear.[1] The next statement creates the type Emp_t as a subtype of Person_t, adding the attribute salary. Looking ahead for a moment, the final data definition statement in Figure 1(b) adds a second additional attribute, dept, to the type Emp_t. This attribute is a *reference* attribute that refers to (i.e., uniquely identifies) an object of type Dept_t; its definition is deferred until after the type Dept_t is defined because the types Emp_t and Dept_t refer to one another (creating a circularity which the **alter type** statement breaks). The data definition statement following the creation of Emp_t creates type Prof_t as a subtype of Emp_t, adding rank and specialty attributes; an instance of type Prof_t will

[1] The clause **mode db2sql** can be ignored; it protects UDB Version 5.2 applications from future changes that could occur in the SQL99 standard before it is finalized and published in late 1999.

```
create table person  of Person_t
             (ref is oid user generated);
create table emp     of Emp_t     under person
             inherit select privileges;
create table prof    of Prof_t    under emp
             inherit select privileges;
create table student of Student_t under person
             inherit select privileges;

create table dept    of Dept_t
             (ref is oid user generated,
              mgr with options scope emp);

alter table emp alter column dept add scope dept;
```

Figure 2: DDL for creating a table hierarchy

thus have a total of six attributes: name, birthyear, salary, dept, rank, and specialty. The fourth statement defines one last subtype of Person_t, namely Student_t. These four structured types, Person_t, Emp_t, Prof_t, and Student_t, together form the Person_t *type hierarchy*. The final type definition in the figure defines the type Dept_t. Note that this type also contains a reference attribute, mgr, which references an object of type Emp_t.

Given these types, we now need places to store their instances. Like rows in the relational world, typed objects reside in tables defined within a DB2 database. UDB supports a variation of the SQL **create table** statement that creates a *typed table*. To store instances of subtypes, one creates typed *subtables* as well; one can create multiple typed tables of a given type if desired. To provide homes for objects of the types defined above, we could write the table definitions of Figure 2.

The first definition in Figure 2 creates a typed table (or object table) named person that can hold Person_t objects. Object table definitions are required to specify a column name that can be used to refer to the object id of their contained objects; thus, the person table will appear to have three columns, an *object id* column called oid (the name specified in the **ref is** clause of the table definition) plus one column for each attribute of Person_t (name and birthyear). The phrase **user generated** tells the system that the object id values for objects in this table will be provided by the user when objects are initially inserted into the table.[2] Object id values must be unique within the table plus all of its supertables and subtables; the system enforces this requirement at insert time. Object id values can only be provided at insert time, when an object is initially created; the object id associated with an existing object is considered immutable and is thus not updatable.

The next three definitions create *subtables* of the

[2] The alternative would be **system generated**, in which case the system would be expected to automatically generate an object id for each inserted row.

person *root table*; emp and student are immediate subtables of person, and prof is a subtable of the emp table. Subtables inherit the columns of their supertables, so no object id column is specified in those definitions; they inherit their oid column from the root table person. Each subtable's type must be an immediate subtype of its supertable's type, and a given table can have only one subtable of any particular type. The person table and subtables in the figure, taken together, form the person *table hierarchy* and are closely related by the "substitutable" DML behavior that we will describe in the next section. They are also managed as a unit for certain purposes, e.g., when certain administrative commands and utilities are invoked (against the root table). For most purposes, a good mental model for a table hierarchy is to think of its root table (e.g., person) as essentially being a heterogeneous collection of objects of its underlying type (e.g., Person_t objects) and subtypes thereof (e.g., Emp_t, Prof_t, and Student_t objects). The clause **inherit select privileges** in each subtable creation statement tells the system that users who hold select privileges on the root table (person) when the subtables are created should be granted those same initial privileges on the subtables. The final table creation statement above creates a separate typed table named dept to hold Dept_t objects; it too has an object id column that we have chosen to call oid.

One other data definition feature that is very important in our SQL DDL extensions is the notion of reference *scope*. The **create table** statement for the dept table contains a clause of the form "mgr **with options scope** emp". This clause tells the system that the Emp_t objects referred to from the reference column mgr of the dept table will reside in the emp table or any subtable thereof, e.g., the prof table in our example. (The **with options** clause is a new addition to the **create table** statement in UDB. It is needed to provide an opportunity to specify any table-specific properties for columns that arise from attributes of the table's type; reference scopes are a very important example of such a property.) Similarly, the last DDL statement in the figure, the **alter table** statement, tells the system that Dept_t objects referred to from the reference column dept of the emp table (and its subtables) will reside in the dept table.

Scope information is used by the system when processing queries involving the dereference operator (->) discussed in the next section. As we will discuss later, scope information is used both for performance reasons, to facilitate query optimization, and for authorization reasons, to allow static authorization checking for queries that involve dereferences. It should be noted that scopes are not a substitute for referential integrity; scopes simply provide information (for dereferencing) about the intended target table of a reference column. Referential integrity (i.e., prevention of dangling references) can be supported for reference columns via UDB's pre-existing referential integrity enforcement facilities, which work for columns of any type. Scopes and referential integrity have been kept orthogonal to allow users to choose whether or not to pay the performance price of referential integrity; some applications inherently ensure it, making any extra checking overhead redundant and undesirable. In the future, we plan to infer scopes from referential integrity constraints (but not vice versa) when possible for reference columns.

3 SQL Data Manipulation Language Extensions

The DDL extensions just described have a set of corresponding DML extensions. In particular, the basic SQL DML statements—**insert**, **select**, **update**, and **delete**—have been extended to deal with typed table hierarchies, and path expression support has been added to the language to enable convenient and natural traversal of object references (a la GEM [16]). The **insert** statement, when applied to a table or subtable, creates a new typed object in the specified table or subtable and initializes its attributes using the values provided by the **insert** statement. The **select**, **update**, and **delete** statements, when applied to a table or subtable, operate on the requested attributes from the target table or subtable *and all of its subtables*—that is, they treat subtable rows as being *substitutable* for supertable rows. If the all-columns operator, *, is specified, the returned attributes are those defined at the targeted table or subtable's level of the table hierarchy. Similarly, all columns mentioned by name in a query that targets an object table or subtable must be defined at (or above) the targeted table's level of the type hierarchy. For path expressions, an arrow operator analogous to that of C++ is provided, and path expressions involving one or more uses of this operator can appear just about anywhere a value expression is permitted in SQL. Finally, we have also added features to SQL to facilitate the manipulation of objects based on their runtime type. These features are best illustrated via a series of examples.

To add a new Emp_t object to the database with oid o100, name Smith, birth year 1968, and salary $65,000, assigning the new employee to work in the CS department, we would use an SQL **insert** statement to create the employee in the emp subtable of the person table:

```
insert into emp (oid, name, birthyear, salary, dept)
  values (Emp_t('o100'), 'Smith', 1968, 65000,
          (select oid from dept where name = 'CS'));
```

The object id for the object created above is provided by typecasting a Varchar constant into a **Ref**(Emp_t) value (because references are strongly typed). The cast is accomplished using a cast function that the system automatically generates when a new structured type is

created; again, the system will check to ensure that the newly inserted object has an object id that is unique within the person table hierarchy. Finally, notice that the new employee's department reference is obtained using a subquery that selects the object id of the desired department.

As mentioned above, the **select**, **update**, and **delete** statements all operate on table hierarchies in a manner that is based on the principle of substitutability (or equivalently, on the mental model of heterogeneous collections of objects). Thus, for example, we could select the oid, name, birth year, salary, and department reference of employees of all types (i.e., Emp_t and/or Prof_t objects) born after 1970 who earn more than $50,000 per year via the following query:

```
select  E.*
from    emp E
where   E.birthyear > 1970 and E.salary > 50000;
```

Similarly, we could change the birth year for the person (who might happen to be a regular person, an employee, a professor, or a student) whose oid is o200 to be 1969 via an **update** statement:

```
update person P
set    P.birthyear = 1969
where  P.oid = Person_t('o200');
```

Since this statement targets the person table, it can only mention columns defined at the person level of the table hierarchy (e.g., it cannot mention Emp_t, Student_t, or Prof_t columns). Finally, we could delete all employees (both regular and professors) who earn too much money via:

```
delete from emp E where E.salary > 500000;
```

Of course, to execute these statements, the user must have the proper SQL authorizations. UDB requires explicit authorization on the statement's target subtable; to perform a **delete** on the emp table, the user would have to hold the **delete** privilege there. It is possible (and sometimes desirable) to grant different privileges at different levels of a table hierarchy. As a result, the person table creator might grant a full set of privileges to some user, but that user will not be able to explicitly operate on the emp subtable by virtue of holding person privileges. Instead, the emp subtable creator would have to decide which privileges to give out in order to protect the attributes (e.g., salary) introduced at the emp level of the person table hierarchy.

UDB's support for path expressions greatly simplifies queries that select attributes from a set of related objects by permitting relationships to be explicitly traversed using the dereference operator –>. For example, to find the employee name and salary, as well as the corresponding department name and budget, for all employees who work in departments that have budgets that exceed $150,000 per person, we could simply say:

```
select  E.name, E.salary, E.dept->name,
        E.dept->budget
from    emp E
where   E.dept->budget > 150000 * E.dept->headcnt;
```

In the case where a qualifying employee has no department (because its dept reference attribute is null or dangling), the path expression yields null (a la GEM [16], and unlike OQL [5], which would raise a user-unfriendly runtime exception). Path expressions are similar in this regard to left outer joins. As another example of how path expressions can simplify a query, we could find the names of all of the employees whose manager's manager is Jones, which would require writing a five-way join query in the absence of path expression support, by simply saying:

```
select  E.name
from    emp E
where   E.dept->mgr->dept->mgr->name = 'Jones';
```

In addition to the aforementioned extensions, UDB also makes it possible to restrict a query's attention to objects of a particular type (or types) and to inquire about an object's type. For example, to select the oid, name, birth year, salary, and department reference of employees who are *exactly* of type Emp_t (i.e., objects that reside in the emp table, not a subtable of emp) and who work in a department with a budget of more than $10M, we could say:

```
select  E.*
from    only(emp) E
where   E.dept->budget > 10000000;
```

We expect this to be the most commonly used form of type restriction, which is why it has a special syntax (**only**). For more general cases, UDB supports a *type predicate* in its dialect of SQL. The type predicate compares the runtime data type of a structured type instance (obtained by **deref**erencing a reference value) with a list of types and returns true if its runtime type is one of those in the list. As an example, we could use a type predicate to select the oid, name, and birth year of people born before 1965 who are either of type Student_t or else exactly of type Person_t:[3]

```
select  P.*
from    person P
where   P.birthyear < 1965 and
        deref(P.oid) is of (Student_t, only Person_t);
```

Finally, one other SQL extension that UDB provides is the ability to query the **outer** union of a table hierarchy. For example, the next query selects the type name, object id, and all possible attributes of the lucky employee whose oid is o013. By all possible attributes, we mean all attributes that an employee object *might* have (as an instance of Emp_t or its subtype Prof_t) depending on its runtime type. The **outer** union returns

[3] The reader can think of this example's type predicate as "... **and** P **is of** (Student_t, **only** Person_t) ...", which is the syntax we would have preferred. Unfortunately, we were unable to find a way to make this nicer syntax acceptable in the full context of SQL99.

null values for inapplicable attributes (e.g., for employees who are not professors, rank and specialty will be null). Duplicate attribute names within the hierarchy, if they arise, can be disambiguated using SQL's **from** clause column renaming feature (**as**). The example query is:

```
select  type_name(deref(E.oid)), E.*
from    outer(emp) E
where   E.oid = Emp_t('o013');
```

The type_name function is similar to the type predicate, but instead of testing the runtime type of an object, it returns the runtime type name. (There is also a type_schema function that returns the name of the schema in which the runtime type resides, and a type_id function that returns the type's database-specific internal id.) Given an object id, this form of query is especially useful for obtaining all of the data associated with the referenced object, including its runtime type, through a single call to a dynamic query API like ODBC or JDBC.

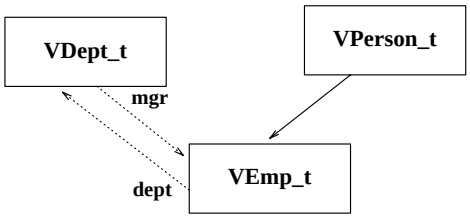
4 Advanced SQL Data Definition Language Extensions

As described in the introduction, relational database systems have a number of advanced features that users have come to rely on for providing alternative views of their base data and expressing business rules and logic. As argued in [14, 10], object-relational database systems must fully support such features, as otherwise they will be a step backwards in some important ways. This section describes how UDB addresses this requirement (plus several other DDL requirements).

4.1 Object Views and View Hierarchies

In relational databases, views are virtual tables whose contents are defined by a query; to a user's application or query, a view looks just like a table. In UDB, we support *object views* and *object view hierarchies* that provide this same transparency and flexibility for users of typed tables and table hierarchies. In particular, we support the creation of typed object views, and these views can either be root views or subviews of other object views. The body of an object view is a query whose select list is type-compatible with the declared type of the view. As prescribed in a seminal paper on views of object databases [9], UDB supports networks of object views that reference one another to form view schemas. UDB's object view facility was influenced by the Garlic object-centered view concept [4], which in turn was influenced by prior work on object views.

Again, we will explain this UDB feature using an example. Suppose that we wished to construct a set of interrelated object views that could be shown to users instead of the base tables and subtables defined earlier. Further suppose that we only wished to include non-academic employees and well-funded departments



(a) Schematic of view entities and relationships

```
create type VDept_t as (
    name Varchar(20)
) mode db2sql;

create type VPerson_t as (
    name Varchar(40)
) mode db2sql;

create type VEmp_t under VPerson_t as (
    dept Ref(VDept_t)
) mode db2sql;

alter type VDept_t add attribute mgr Ref(VEmp_t);
```

(b) DDL statements for creating the view types

Figure 3: A hierarchy of view types

(those with a budget greater than $1M) in our views. UDB's object views are based on the same type system as its regular object tables. Figure 3(a) depicts a set of view types, and Figure 3(b) gives a set of type definitions to create these view types. Except for the missing attributes, these look similar to our previous type definitions. It is important to notice, however, that these types are interrelated among themselves: the dept attribute of type VEmp_t is of type **Ref**(VDept_t), and the mgr attribute of type VDept_t is of type **Ref**(VEmp_t).

Given these type definitions, we can create the desired object view hierarchy as shown in Figure 4. The first two object view definitions are similar. Each defines a typed view in much the same way that typed tables were defined earlier, and then each provides a query that selects the appropriate set of view objects. Note that objects in an object view have object ids that are created by typecasting their base object ids to be view object ids. (The intermediate cast to Varchar is required because a reference to one type cannot be cast to be a reference to an unrelated type without violating strong reference typing.) The third definition creates a subview vemp of the object view vperson. The query associated with the subview selects the same first two columns as its parent view (subtyping the object id appropriately), extending them with the additional attribute that VEmp_t instances have as compared to VPerson_t objects. In UDB V5.2, all object views in an object view hierarchy are required to be defined over the same underlying table or table hierarchy, with the same column being used as the basis

```
create view vdept of VDept_t mode db2sql
    (ref is oid user generated)
    as select VDept_t(Varchar(oid)), name,
              VEmp_t(Varchar(mgr))
       from only(dept)
       where budget > 1000000;

create view vperson of VPerson_t mode db2sql
    (ref is oid user generated)
    as select VPerson_t(Varchar(oid)), name
       from only(person);

create view vemp of VEmp_t mode db2sql
    under vperson inherit select privileges
    (dept with options scope vdept)
    as select VEmp_t(Varchar(oid)), name,
              VDept_t(Varchar(dept))
       from only(emp);

alter view vdept alter column mgr add scope vemp;
```

Figure 4: DDL for creating a view hierarchy

for the view's object id column, and the sets of objects identified by the body (query) of each view/subview are required to be disjoint. These rules are enforced by UDB at subview definition time to ensure that the contents of views/subviews in a view hierarchy have the same logical properties as do tables/subtables in a table hierarchy. Finally, note that reference columns of object views are scoped, just like reference columns of object tables, and that their scopes can be other object views. (The dept column of the vemp view has vdept as its scope in our example, and the reference column vdept.mgr has vemp as its scope).

Once defined, object views can be queried, used in path queries, and even updated if their defining **select** statements are of an updatable nature (which they are in our example). For instance, the following query finds the names and department names of view employees who work in a view department whose name starts with D; professors will not be considered due to the nature of the vemp view definition, and departments with small budgets will be filtered by the vdept view definition (thereby making references to such departments behave like dangling references, returning null values in the context of the query's path expressions). Because of the scopes given in the view definitions, the SQL expression dept–>name is a path expression that follows references from vemp.dept to get the corresponding vdept.name values:

```
select  E.name, E.dept->name
from    vemp E
where   E.dept->name like 'D%';
```

Type predicates can be used with object views as well, as shown by the following query for finding the names of Scottish people who are regular (i.e., non-professorial) employees of the university:

```
select  P.name
from    vperson P
where   deref(P.oid) is of (only VEmp_t) and
        P.name like 'Mc%';
```

While our example showed object views of object tables, it is important to note that UDB's object view facilities can also be used to create object views and view hierarchies out of existing relational (i.e., non-object) tables. This provides an important migration path for users who have legacy relational data but wish to begin exploiting object-relational modeling in new applications. To provide even better support for such users, we have recently added (but not yet shipped) a new clause, **ref using**, that can be added to a **create type** statement to direct UDB to use a specified data type to represent object ids for that type and its subtypes. This is useful when creating object views of legacy tables, as different tables often use different primary key types. We have also recently relaxed some of the V5.2 restrictions on object view definitions (e.g., so that view hierarchies can be defined over multiple legacy tables).

4.2 Constraints, Triggers, and Other Features

As mentioned earlier, it is important that support for constraints and triggers be extended to the object-relational world as well. To be consistent with the inheritance model that table hierarchies imply, such features must be definable on tables or subtables within a hierarchy and they must be inherited by any subtables of the table upon which they are defined. In UDB, support is provided for defining **not null** and **unique** constraints, as well as for providing **default** values and defining indexes, on tables in a table hierarchy. One can define a **not null** constraint on a column at the point in the table hierarchy where it first appears (i.e., when defining the subtable that introduces the column into the hierarchy), and **default** values and non-unique indexes can be specified for columns at that same point. UDB also supports **unique** constraints and unique indexes (but presently only permits them to be defined on the root table of a table hierarchy). In all cases, these features are implicitly inherited by subtables; the corresponding columns of subtables are thus subject to the same rules and indexing. For example, if the emp subtable in our original person table hierarchy included a **not null** constraint on its dept column, this constraint would also be enforced for rows of its prof subtable.

Support for more general constraint and trigger inheritance is presently under development at IBM Almaden. We have **check** constraints working on table hierarchies, and **foreign key** constraints are in progress. As an example, it is now possible for the emp subtable in our person table hierarchy to include the constraint "**check** (salary > 0)" to ensure salary validity, and it will be possible shortly to have a constraint of the form "**foreign key** (dept) **references** dept(oid)" to maintain referential integrity for employees' department references. Once defined, these constraints are inherited and enforced for prof as well as

emp rows. Similarly, if the dept table had the constraint "**foreign key** (mgr) **references** emp(oid)", UDB would ensure that every department has a corresponding manager who is an employee (or some subtype thereof, of course, but not just a person or a student). Again, we remind the reader that scope clauses do not make such referential integrity constraints redundant—**scope** clauses provide reference target table information, but they do not by themselves tell the system to actively prevent dangling references.

We are currently working on trigger inheritance at Almaden as well. A trigger defined on a supertable will be automatically inherited by its subtables and then fired whenever a triggering modification occurs to either the table upon which it is defined or to any of that table's subtables. (It is worth noting that type predicates can be used to limit this behavior in those—rare, we believe—cases where inheritance by subtables is not the trigger definer's desire; of course, the same technique can be used to limit inheritance of check constraints if so desired.)

5 Implementation Issues

In this section, we provide a high-level summary of some of the key design tradeoffs, considerations, and decisions that we faced while designing and implementing these new UDB object-relational extensions. The main principles that guided our thinking and decision-making were:

1. Performance of all features needs to be at least as good as their relational equivalents.
2. The design must be amenable to future work on schema- and instance-level type migration.
3. The bulk of the initial UDB changes should be in the query compiler if possible.
4. Structured type instances must eventually be storable in columns as well as rows of tables.

The first principle almost goes without saying—we felt it would be unacceptable to offer "cool new object-relational features" that caused customer applications to perform worse than equivalent relational solutions. The second principle was a result of looking ahead at some "must have" functionality that we did not have time to deliver in our first release, but which we knew would be critical in the not-too-distant future. The third consideration was motivated by a desire to localize our changes as much as possible, at least initially, and to get as much functionality "for free" as we could. Essentially, we wanted UDB's indexing mechanisms, query rewrite technology, query optimizer and supporting statistics, parallel query execution algorithms, and so on, to work for data in table hierarchies with as few changes as possible. The fourth principle was another future thought; we wanted to avoid making any decisions that would somehow preclude structured type column values in the long run.

5.1 Representing Table Hierarchies

An early question that we faced was how to physically represent table hierarchies. In light of our third design principle, we seriously considered three possible storage alternatives. (We refer the interested reader to [15, 11] for other analyses of storage options for systems with type hierarchies.) Note that in each case, the implementation decision would be invisible to end users; e.g., regardless of UDB's internal storage method, users would see our example person table and its emp, prof, and student subtables as separate (but related) typed tables.

One approach that we considered was the *hierarchy table* approach, where each table hierarchy (as a whole) corresponds to one physical implementation table under the covers; this table contains the union of the columns required to store rows of any subtable in the hierarchy. For example, the hierarchy table for the person table hierarchy would contain a type tag column (to distinguish among rows of different subtables), an object id column, the name and birthyear columns for person rows, the additional salary and dept columns for emp rows, the rank and specialty columns for prof rows, and the major and gpa columns for student rows. Any given row would have a type tag indicating whether it is a Person_t, Emp_t, Prof_t, or Student_t row, and the inapplicable columns for a given row would simply contain null values.

The second approach that we considered was *vertical partitioning*. This approach would have one physical "delta table" for each table in the table hierarchy. The physical person table would contain a type tag and and object id plus the person name and birthyear attribute values. The physical emp table would have an object id column plus just the additional ("delta") attributes of employees, i.e., salary and dept. The physical prof table would contain three columns, an object id plus rank and specialty. The physical student table would contain three columns as well, namely, oid, major, and gpa. With this approach, an object of type Prof_t would be physically spread over the person, emp, and prof delta tables (with its parts being linked by their oids).

The final table hierarchy storage option that we considered using was *horizontal partitioning*. This approach would also utilize one physical table for each table in the table hierarchy, but in this case each physical table would contain all of the columns for rows of that table of the table hierarchy. For example, the prof table would have a total of six columns: oid, name, birthyear, salary, dept, rank, and specialty. No type tag column would be needed since all the instances of each type present in the table hierarchy would be

stored together in a separate physical table; the type of each row is implied by the particular storage table that it resides in.

After studying the pros and cons of the three alternatives, we rejected the vertical partitioning approach because of the joins required to fully materialize a row of a subtable—we were afraid that they would make query performance unacceptable, violating our first guiding principle. (Similar reasoning caused GEM's designers to reject this alternative as well [15].) In addition to query performance concerns, we anticipated problems supporting multi-column constraints and multi-column indexes under the vertically partitioned scheme. For example, a simple check constraint involving both inherited and non-inherited columns could not be checked without extending UDB to do joins during constraint checking; teaching UDB's storage manager to index columns split across multiple tables would have been even more of a challenge.

We also rejected the horizontal partitioning approach for UDB, though not as quickly. One reason for rejecting horizontal partitioning was the difficulty that would arise in checking the uniqueness of user-provided object id values (or other unique-constrained columns) across subtables within a table hierarchy, as we would have to ensure their uniqueness across multiple physical tables. Perhaps more seriously, we were also concerned about the costs that this approach would imply for small lookups or joins of table hierarchies. For example, without a multi-table indexing method, a simple query to find the person named Codd would imply four physical lookups (person, emp, prof, student). Similarly, a query to find pairs of people born in the same year, joining person with itself, would internally require either joining two four-way unions or performing all individual pairwise joins of the four underlying physical tables and unioning the results. We were worried that such situations would be fairly common and would lead to poor query performance, again violating our first guiding principle.

As a result of this analysis, we settled on the hierarchy table approach for storing table hierarchies in UDB. We were convinced that the hierarchy table alternative would provide the best initial performance with the fewest problems and restrictions. Since each row contains all attributes (both inherited and non-inherited), no joins are needed to assemble object instances for queries or constraints; indexing combinations of inherited and non-inherited columns poses no problem either. Since all objects in a table hierarchy live in the same physical table, it is easy to properly enforce unique constraints on oids and user-specified columns. A query that selects an object from a table hierarchy maps to a simple lookup in one index on the hierarchy table, and a query that joins a pair of hierarchies together maps to a simple two-way physical join—which nicely satisfies both our first and third design principles. The hierarchy table approach also simplifies the migration issues that are the focus of our second design principle—migrating an object from one type to another within a table hierarchy becomes a simple type tag update under the covers, and modifying a type can be accomplished at the physical level via an **alter table** operation on the affected hierarchy tables. Thus, the hierarchy table approach provided an expedient path to having a fully functional first implementation of table hierarchies that met our design goals. The main downside of the approach is its potential tuple width, as the width of a hierarchy table's rows is a function of the size of the hierarchy rather than of its individual types. However, null values can be represented efficiently in modern data managers [12], and UDB already handles nulls efficiently for variable-length columns (which tend to be the largest columns), so this drawback seemed less serious than those of the other storage approaches.

To quantify some of the tradeoffs discussed above, we have conducted a small set of preliminary experiments comparing the hierarchy table approach to the horizontal and vertical partitioning alternatives. We constructed a three-level type hierarchy with a root type, two subtypes of the root, and two subtypes of each intermediate type (yielding seven types in all). We constructed a corresponding table hierarchy with 40,000 rows of each type; the overall database contained 280,000 rows and constituted approximately 64MB of data. The root type had an integer attribute plus a 200-byte padding attribute; each additional subtype added another integer attribute to those of its supertype. We stored this data using UDB V5.2's implementation approach (the hierarchy table) as well as in sets of relational tables modeled after the other two alternatives. The integer attributes and oid attributes were all indexed, the data was loaded top down by type, and we gathered optimizer statistics for all three approaches before running our tests.

Figure 5 shows the results from running seven different queries of interest against our test database. The test platform was an IBM ThinkPad 770 machine with 128MB of memory running DB2 Version 5.2 under Windows NT. Space precludes a careful analysis of all of the queries and results; hopefully the reader will find them fairly self-explanatory. The root-level queries operate on the root of the table hierarchy and access rows of the root table and all subtables, while the leaf-level queries operate on a leaf table and access its rows only. From the results, it is evident that vertical partitioning pays a price due to joins for the leaf queries, while horizontal partitioning pays a price due to unions for the join queries. The sums of the times at the bottom of the table, while not especially meaningful in absolute terms, suggest that the hierarchy table approach has the most stable overall performance characteristics.

Query	Hierarchy Table	Vertical Partitioning	Horizontal Partitioning
1. count all rows (root)	1.70 sec	1.92 sec	2.16 sec
2. select 1 row (root)	0.27	0.26	0.25
3. select 1 row (leaf)	0.20	0.25	0.18
4. select 1 row and join (root)	0.22	0.27	24.48
5. select 1 row and join (leaf)	0.20	0.33	1.98
6. join all rows (root)	22.75	15.54	86.72
7. join all rows (leaf)	8.51	39.63	8.87
sum of 1–7	33.85 sec	58.20 sec	124.64 sec

Figure 5: Performance comparison of storage alternatives

5.2 References and Path Expressions

One of the major facets of our UDB extensions is support for references and path expressions. Semantically, a path expression is similar to a subquery. For example, consider:

```
select  E.name, E.dept->name
from    emp E
where   E.salary > 90000;
```

If the system knows that the scope of emp.dept is the dept table, this is essentially equivalent to:

```
select  E.name, (select D.name
                 from   dept D
                 where D.oid = E.dept)
from    emp E
where   E.salary > 90000;
```

The subquery, like the corresponding path expression, returns the name of the matching department if there is one, returning null otherwise.

As we mentioned earlier, UDB requires a reference to have a scope if it is dereferenced. One reason for this was the (lack of) performance observed for unscoped references in the BUCKY benchmark [2]. In particular, when a path expression appears in a predicate, knowing the target table for a reference enables the system to fully optimize the query as a join rather than performing naive pointer-chasing. (The latter is what happened to a target system of the BUCKY work; it is also how many object database systems process queries in the absence of type extents or path indices.) By simply requiring references to be scoped, we avoid this problem and ensure that we will be able to answer users' queries efficiently (a la guiding principle number one). In addition, scope information enables us to check authorizations statically for path queries, as we can always identify the tables involved in a query at compile time rather than waiting until runtime to (more slowly, on a row-by-row basis) check whether or not the user has the appropriate authorizations.

It is worth noting that having scopes at the schema level allows reference values to be kept relatively small in size and simplifies the implementation of user-defined references. With scopes, stored reference values do not need to contain table names—they must simply contain enough information to uniquely identify a row within a given table hierarchy, with query compilation ensuring (via an internal type-tag predicate) that only rows of the targeted table/subtables are actually picked up by a dereference operation. Keeping references small is beneficial for keeping the size of complex object-relational databases reasonable. Not storing type or subtable information in references has another very important advantage as well—it means that UDB will have no trouble efficiently supporting type migration (e.g., promoting a Person_t object to be an Emp_t), whereas systems that place such information in references will have to track down all affected references (if that is even possible) and update their target type or subtable indicators.

Given scope information, a possible approach for implementing path expressions would be to simply translate each one internally into an independent subquery. However, there are several problems with this approach. First, it is inefficient, particularly in cases where there are similar path expressions in a given query (e.g., if a number of dept attributes had been requested in the path query above). Second, if the query is run at a non-serializable level of consistency (e.g., cursor stability), independent subqueries could produce surprising results in the face of concurrent updates. As a result, we first translate path expressions internally into a special form of shared subqueries. UDB's query rewrite component then translates these shared subqueries into outer joins when possible; moreover, it rewrites them into inner joins in many cases, such as when a given path appears in the query's **where** clause. From there, UDB's query optimizer is free to consider all of its usual join orders, join methods, parallel execution options, and so on.

Also on the topic of references, UDB V5.2 supports only user-generated object id values, though we have prototyped system-generated object ids as well. There are several reasons for our decision to ship user-generated object ids first. One factor was a series of discussions with a UDB customer who wanted to migrate from an object mapping layer that they had implemented on top of a pure relational system to exploit the (then) forthcoming UDB object support. That

customer already had a number of existing "legacy" objects, with existing ids that appeared externally in operating system files as well as in databases. The prospect of being forced to re-identify all of their legacy objects posed a problem for them. Another consideration was our desire to support the efficient initial loading of object-relational data in cases where the user has a convenient way to generate object ids outside of UDB. In the BUCKY benchmark, the object-relational load times were an order of magnitude worse than relational load times [2] because each object that contains system-generated references had to be connected (joined) to the objects that it refers to, and these connections cannot be finalized until their object ids have been generated by creating the objects themselves. In contrast, user-generated object ids allow the loading of data from files—including references—into typed tables at full relational speeds. A related consideration was database creation in external object caches. We wanted to provide efficient support for applications where a graph of objects is created externally, in a cache, and then handed to the system. If the only way to generate object ids is for the system to do it as objects are inserted, this process becomes messy (e.g., one must topologically sort the cached object graph) and/or expensive (because one must backpatch object references between objects after insertion). UDB will of course support system-generated object ids as well in the future, but we have come to believe that user-generated object ids are in fact preferable in a number of common situations.

6 Relationship to Other Work and Systems

As alluded to earlier, our work has been influenced by a number of previous papers and systems (too numerous to cite and do justice to here). The seminal work on GEM [16, 15] heavily influenced our model for path expressions as well as some of our thinking with respect to hierarchy storage. We were also heavily influenced by past experiences in the University of Wisconsin EXODUS project [3] as well as those from a number of other projects from the same era [1] and by various "manifestos" on next-generation database system requirements [14, 10].

It is also appropriate to compare UDB to other vendors' systems; we do so very briefly here. Both Informix and Oracle offer object-relational features as well. Informix supports user-defined structured types and hierarchies of tables; however, to the best of our knowledge, Informix does not yet include support for references, path expressions, or object views. Oracle 8 supports user-defined structured types and object views, but does not provide any support for inheritance or table/view hierarchies. Informix and Oracle 8 both provide degrees of support for methods, nesting of structured types, and collection-valued attributes, features not yet provided in the released version (V5.2) of UDB. Some of the unique aspects of UDB are its support for user-generated object ids, its aggressive approach to scopes and consequently to path query optimization, and its unique support for object views including view hierarchies. Method and nested structured type support for UDB are working in the lab (specifically, IBM's Santa Teresa Laboratory), and we are currently exploring collection type support at IBM Almaden.

7 Status and Future Plans

Most of the object-relational features that have been covered in this paper are available today in Version 5.2 of the DB2 UDB product. These features include structured types, object tables, type and table hierarchies, references, path expressions, and object views and view hierarchies. These features are available on all supported V5.2 platforms, which include a wide variety of operating systems (most common variants of Unix, NT, Windows95, and OS/2) and a variety of serial and parallel (SMP and MPP) hardware platforms. Thus, DB2 UDB now provides a solid initial foundation for the management of complex object data.

We are currently extending this work in several ways. As mentioned earlier, we have check constraints working on table hierarchies and are completing our work on referential integrity and triggers for table hierarchies. Near-term things that we plan to address next include type migration, type evolution, and system-generated oid support. Other topics of current interest include support for collection types, e.g., collection-valued attributes a la ODMG [5], and ways to connect object-relational data to the web using XML or extensions thereof.

Acknowledgments

The authors wish to thank a number of IBM researchers and DB2 UDB developers who helped in one way or another to make tables of objects a reality. Hugh Darwen and Stefan Dessloch provided consulting on various SQL99 language design issues. Cheryl Greene has been a source of moral support and guidance. Discussions with Peter Schwarz, Ed Wimmers, and Jerry Kiernan influenced our thinking on object views. Hamid Pirahesh, Bobbie Cochrane, Richard Sidle, George Lapis, Cliff Leung, Jason Sun, and others helped us to understand and exploit the existing UDB query processing infrastructure; they also made minor extensions to simplify our task. Gene Fuh, Brian Tran, and Michelle Jou made contributions as part of the DBTI team. Paul Bird provided expertise on a number of technical issues. Walid Rjaibi and Calisto Zuzarte made the changes necessary to educate the UDB optimizer statistics utilities about table hierarchies. Leo Lau added table hierarchy support to UDB's data ex-

change utilities. CM Park, Carlene Nakagawa, and Raiko Nitzsche helped significantly with system testing. Finally, discussions with our Berkeley friends Mike Stonebraker, Joe Hellerstein, and Mehul Shah helped to improve the presentation.

References

[1] M. Carey and D. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings VLDB Conference, Mumbai (Bombay), India*, pages 3–14, 1996.

[2] M. Carey, D. DeWitt, J. Naughton, et al. The BUCKY object-relational benchmark. In *Proceedings SIGMOD Conference, Tucson, Arizona*, pages 135–146, 1997.

[3] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. In *Proceedings SIGMOD Conference, Chicago, Illinois*, pages 413–423, 1988.

[4] M. Carey, L. Haas, P. Schwarz, et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings IEEE RIDE–DOM Workshop, Taipei, Taiwan*, pages 124–131, 1995.

[5] R. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[6] E. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[7] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[8] L. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, 1990.

[9] S. Heiler and S. Zdonik. Object views: Extending the vision. In *Proceedings ICDE, Los Angeles, California*, pages 86–93, 1990.

[10] W. Kim. Object-oriented database systems: Promises, reality, and future. In *Proceedings VLDB Conference, Dublin, Ireland*, pages 676–687, 1993.

[11] B. Nixon et al. Implementation of a compiler for a semantic data model: Experiences with Taxis. In *Proceedings SIGMOD Conference, San Francisco, California*, pages 118–131, 1987.

[12] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1997.

[13] M. Stonebraker. *Object-Relational Database Systems: The Next Great Wave*. Morgan Kaufmann, 1996.

[14] M. Stonebraker et al. Third-generation database system manifesto. *ACM SIGMOD Record*, 19(3):31–44, 1990.

[15] S. Tsur and C. Zaniolo. An implementation of GEM – supporting a semantic data model on a relational back-end. In *Proceedings SIGMOD Conference, Boston, Massachusetts*, pages 286–295, 1984.

[16] C. Zaniolo. The database language GEM. In *Proceedings SIGMOD Conference, San Jose, California*, pages 207–218, 1983.