Implementation of SQL3 Structured Types with Inheritance and Value Substitutability

You-Chin (Gene) Fuh, Stefan Dessloch, Weidong Chen*, Nelson Mattos, Brian Tran
Bruce Lindsay[1], Linda DeMichiel[2], Serge Rielau[3], Danko Mannhaupt

IBM Santa Teresa Laboratory, [1]IBM Almaden Research Center
[2]Sun Microsystems, [3]IBM Toronto Laboratory

Abstract

SQL3 has introduced structured types with methods and inheritance through value substitutability. A column of a structured type in a relation may contain values of the structured type as well as values of its subtypes. Integrating structured types with the existing database engine raises some interesting challenges. This paper presents the DB2 approach to enhance the IBM DB2 Universal Database (UDB) with SQL3 structured types and inheritance. It has several distinctive features. First, values of structured types are represented in a self-descriptive manner and manipulated only through system generated observer/mutator methods, minimizing the impact on the low level storage manager. Second, the value-based semantics of mutators is implemented efficiently through a compile-time copy avoidance algorithm. Third, values of structured types are stored inline or out-of-line dynamically. This combines the usability and flexibility with the performance of inline storage. Experimental results demonstrate that the DB2 approach is more efficient in query execution compared to alternative implementations of structured types.

*Contact author: Weidong Chen, IBM Santa Teresa Laboratory, 555 Bailey Ave, Room C347, San Jose, CA 95141. Email: cwd@us.ibm.com. On sabbatical leave from Southern Methodist University.

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

1 Introduction

The relational model [3] has revolutionized the information system world by providing a simple, high-level data model and a declarative query interface. The value-based, declarative nature of the relational model offers high level data independence where the physical organization of the data, including storage and index structures, is separated from the logical schema (or tables) of the data. This has led to modern relational database systems with expressive SQL queries, sophisticated query optimization and execution strategies.

Emerging database applications require scalable management of large quantities of new and complex data together with traditional business data and flexible and efficient querying capabilities for business intelligence. To meet the market demands, object-relational databases have evolved and incorporated various "object" features into the relational database technology, such as user-defined structured types, methods and inheritance [7]. These concepts have been included in the SQL3 standard [6].

A structured type in SQL3 consists of a name, a set of attributes, and a set of methods. Attributes of a structured type are accessed and mutated only through system generated observer and mutator methods. Structured types can be nested and one structured type may be a subtype of another. Inheritance is achieved through a principle called *value substitutability* in the sense that values of subtypes are accepted wherever values of their supertypes are valid.

There are two main places to store values of structured types in an object-relational database: rows in a table and values inside a table column. Given a structured type, a table of that type can be created, where each attribute of the structured type becomes a column of the typed table. Each row of a typed table corresponds to a value of the type. A hierarchy of tables can be defined over a type hierarchy so that standard table operations such as SELECT, UPDATE, and DELETE can be applied to a target table as well as its subtables. This exhibits a form of inheritance where subtables inherit columns from the supertable and operations on the supertable are applicable automatically to all rows in the subtables. The design and the implementation of typed tables in DB2 UDB, including table hierarchies, references, path expressions, and object views, have been addressed in [1].

Values of structured types can also be stored in table columns. A column of a structured type in a table may contain values of the structured type as well as its subtypes, which can be of different sizes. A main challenge of integrating structured types with the existing database engine is to minimize the impact on low level components inside the database engine, and at the same time to support efficient access and manipulation of attribute values of possibly nested structured types with inheritance.

This paper describes our approach to enhance the DB2 UDB with structured types and inheritance through value substitutability. The main contributions are as follows. First, values of structured types are represented in a self-descriptive manner that can be manipulated only through system generated observer and mutator methods. This minimizes the impact on the low level storage manager and provides efficient access and manipulation of structured typed values. Second, the value-based semantics of mutator methods is implemented efficiently through a compile-time copy avoidance algorithm. Third, values of structured types are stored inline or out-of-line dynamically depending upon their sizes. This combines the flexibility of subtyping and nesting of structured types with the performance advantage of inline storage. It should be mentioned that this paper focuses only on the basic infrastructure for supporting structured types within the database engine. Other important issues such as indexing and query optimization involving structured types will be dealt with in a separate paper.

The rest of this paper is organized as follows. Section 2 reviews SQL3 structured types, methods and inheritance. Section 3 discusses several different techniques of implementing structured types and describes the design rationale of our DB2 approach. Section 4 gives a simple compile-time inferencing mechanism that avoids making unnecessary copies of objects for mutator method invocations. Section 5 presents a dynamic mechanism for inline or out-of-line storage of values of structured types. Section 6 compares our DB2 approach with various alternative implementations of structured types and provides performance results on real data sets. Section 7 concludes with a brief summary and some issues for further investigation.

2 SQL3 Structured Types and Inheritance

This section reviews structured types, methods and inheritance in the SQL3 standard [6]. We illustrate these concepts through examples.

2.1 Structured Types with Observers and Mutators

Consider a simple structured type for address:

```
CREATE TYPE Address
AS (street Char(30),
    city   Char(20),
    state  Char(2),
    zip    Integer
   ) NOT FINAL;
```

Every structured type comes with a set of observer/mutator methods for accessing and updating the values of its attributes. For `Address`, the following methods are generated automatically by the system:

- observers: each attribute has a corresponding observer method that returns the value of the attribute given a value of the structured type.

  ```
  Address.street -> Char(30)
  Address.city -> Char(20)
  Address.state -> Char(2)
  Address.zip -> Integer
  ```

  The single-dot notation is for method invocation and is left-associative.

- mutators: each attribute has a corresponding mutator method for updating the value of the attribute.

  ```
  Address.street(Char(30)) -> Address
  Address.city(Char(20)) -> Address
  Address.state(Char(2)) -> Address
  Address.zip(Integer) -> Address
  ```

The implementation of structured types is completely encapsulated in the sense that direct access to its attributes is restricted to observer/mutator methods only.

Values of structured types can be constructed using the "new" notation followed by invocations of the mutators to fill in the attribute values, e.g.,

```
NEW Address().street('555 Bailey Ave')
             .city('San Jose')
             .state('CA')
             .zip(95141)
```

It is also possible for users to define their own constructors (with or without arguments) whose names are the same as the type name.

Structured types can be nested or can be subtypes of one another. For instance, one may define another structured type `ContactInfo` that has an attribute of type `Address`:

```
CREATE TYPE ContactInfo
AS (postal_addr Address,
    home_phone Char(10),
    work_phone Char(10),
    email      Char(20),
    fax        Char(10)
   ) NOT FINAL;
```

Tables can be defined that have columns of structured types, e.g.,

```
CREATE TABLE AddressBook
       (name     Char(30),
        addrinfo ContactInfo);
```

Table AddressBook can be modified using the standard insert/update/delete statements:

```
INSERT INTO AddressBook
VALUES ('Mr.J.Smith',
        NEW ContactAddr()
        .postal_addr(
           NEW Address()
           .street('555 Bailey Ave')
           .city('San Jose')
           .state('CA')
           .zip(95141))
        .home_phone('6502347568')
        .work_phone('4084635678')
        .email('jsmith@us.ibm.com')
        .fax('4084631234'));
UPDATE AddressBook
SET addrinfo.postal_addr.street =
    '123 Almaden Way'
WHERE name = 'Mr.J.Smith';
```

In the update statement above, we are updating the street address for Mr. J. Smith. The semantics of the update statement is equivalent to the following statement:

```
UPDATE AddressBook
SET addrinfo =
    addrinfo.postal_addr(
      addrinfo.postal_addr.street(
                 '123 Almaden Way'))
WHERE name = 'Mr.J.Smith';
```

where the right hand side contains two mutator invocations, one nested inside another. It should be mentioned that mutator methods in SQL3 [6] have a value-based semantics in the sense that they do not have any side effects and simply return values as results. Therefore they cannot be used directly to update a column of a structured type in a table. Instead such a table can be updated only through the standard insert/update/delete statements.

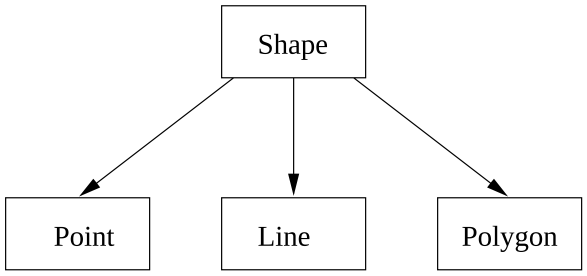

Figure 1: A type hierarchy of shapes

2.2 Inheritance and Value Substitutability

A structured type can be a subtype of another structured type. The subtype inherits attributes and behavior (methods) from its supertypes. Figure 1 shows a simple hierarchy of shapes.

These types can be defined using the CREATE TYPE statement:

```
CREATE TYPE Rectangle
AS (xmin Float,
    ymin Float,
    xmax Float,
    ymax Float
   ) NOT FINAL;

CREATE TYPE Shape
AS (length           Float,
    area             Float,
    mbr              Rectangle,
    numOfPoints      Integer,
    geometry         Blob(1m)
   ) NOT FINAL
METHOD distance(s Shape) RETURNS Float
       LANGUAGE C ...

CREATE TYPE Point UNDER Shape NOT FINAL
METHOD Point(x Integer, y Integer)
       RETURNS Point LANGUAGE C ...

CREATE TYPE Line UNDER Shape NOT FINAL
METHOD Line(x1 Integer, y1 Integer,
            x2 Integer, y2 Integer)
       RETURNS Line LANGUAGE C ...

CREATE TYPE Polygon UNDER Shape NOT FINAL
METHOD Polygon(p1 Point, p2 Point,
               p3 Point, p4 Point)
       RETURNS Polygon LANGUAGE C ...
```

Each structured type may contain a list of method specifications. In the example above, we have a method for computing the distance between two shapes, and a constructor for each of the subtypes of Shape. The implementation of a method is created using a CREATE METHOD statement, e.g.,

```
CREATE METHOD distance(s Shape) for Shape
```

```
EXTERNAL NAME 'shape!distance';

CREATE METHOD Point(x Integer,
                    y Integer) for Point
EXTERNAL NAME 'shape!point';

CREATE METHOD Line(x1 Integer,
                   y1 Integer,
                   x2 Integer,
                   y2 Integer) for Line
EXTERNAL NAME 'shape!line';

CREATE METHOD Polygon(p1 Point,
                      p2 Point,
                      p3 Point,
                      p4 Point)
                      for Polygon
EXTERNAL NAME 'shape!polygon';
```

Values of structured types can be used wherever system predefined types can occur in SQL, including table columns and function/method parameters, e.g.,

```
CREATE TABLE real_estate_info
AS (price    Decimal(9,2),
    owner    Char(40),
    property Shape);

INSERT INTO real_estate_info
VALUES (100000, 'Mr.S.White',
        NEW Point(4,4));

INSERT INTO real_estate_info
VALUES (400000, 'Mr.W.Green',
        NEW Line(5,5,7,8));

INSERT INTO real_estate_info
VALUES (150000, 'Mrs.D.Black',
        NEW Polygon(NEW Point(4,4),
                    NEW Point(6,12),
                    NEW Point(12,12),
                    NEW Point(14,4)));
```

Each row of `real_estate_info` can have a property shape of a different subtype. This is called *value substitutability*, where a column of a structured type can contain values of the type as well as values of all its subtypes.

The principle of *value substitutability* applies not only to table columns but also to function and method parameters. Values of subtypes can be passed to parameters of their supertypes in functions and methods. When a method is invoked, dynamic dispatch is needed to determine the method body to be executed when there is method overriding. In the following query,

```
SELECT owner, price
FROM real_estate_info
WHERE property.distance(Line(10,10,20,20))
      <= 5;
```

if the subtypes of `Shape` define their own `distance` methods, the execution of the query will cause the invocation of a different method for each property value at run time, depending upon the specific shape of each property.

3 Implementation of Structured Types in IBM DB2

We have been implementing structured types with inheritance and method support in the SQL3 standard [6] in the IBM DB2. This section discusses the design rationale of supporting table columns of structured types and presents a self-descriptive representation of values of structured types.

3.1 Design Rationale

From an implementation point of view, there are several different options for integrating values of structured types into table columns. One option is to use some existing datatype to represent values of structured types. Since values of structured types can be of different sizes, due to subtyping and variable-sized attributes, one might use variable-length binary characters or binary large objects. Access and manipulation of structured types can be achieved using user-defined functions and methods. For example, the concept of distinct types in IBM DB2 [2] or opaque abstract data types in Informix [5] can be used to implement this approach. However, since an invocation of a user-defined function or method is an expensive operation, this approach can cause significant overhead for runtime query execution.

Another implementation possibility is to expand a table column of a structured type into multiple columns, one for each attribute of the type. While this may provide efficient access, it becomes complicated for nested structured types or subtypes when every attribute of a structured type is expanded into a separate column. This approach also makes it difficult to support inheritance and value substitutability since subtypes may have additional attributes and values of a subtype can appear anywhere values of its supertypes are valid.

Still another implementation option is to use a separate *side table* to store values of structured types. This approach has been used in geographic information systems such as ESRI's SDE [4]. A difference for structured types would be that the database engine will be aware of such side tables and may be able to optimize queries involving these side tables. Nevertheless the side tables are normally hidden from users, who access attributes of structured types through regular tables that contain other business data. This means that access and manipulation of attributes of a structured type inside a table column require extra joins with the side table, increasing the cost of query optimization and execution.

We have three design goals for the representation and manipulation of values of structured types: (a) to minimize the impact of structured types on the low level storage manager; (b) to support efficient access and manipulation of structured types; and (c) to allow arbitrary nesting and subtyping of structured types.

Our representation of structured types is completely opaque to the storage manager, which sees only a sequence of bytes for a value of a structured type. The representation of structured types is encapsulated and can be accessed or manipulated only through a small set of operations, minimizing the impact of structured types on low level components.

The direct access and manipulation of structured types is possible only through builtin operations for observer/mutator methods, avoiding the extra overhead of user-defined functions. The representation of structured types is self-descriptive and carries its own meta information so that it can be operated upon at run time without requiring any extra information. This also makes it easier to change the representation of structured types for future enhancements.

Values of structured types can be of variable sizes for several reasons. First, the concept of *value substitutability* means that values of subtypes can appear wherever values of supertypes are valid. Subtypes often have attributes in addition to those inherited from supertypes. Second, values of structured types can be mutated, constructing new values of different sizes. Third, a structured type may have attributes that are large objects (called LOBs). To accommodate the variable sizes of values of structured types and to support mutations efficiently, we allow values of structured types to be stored in a non-linearized format in memory. When values of structured types are stored in tables on the disk, the same representation is used except that values of attributes of a structured type will be in a linearized format as they are written to the disk. The use of the same representation avoids any extra conversions when values of a structured type are moved between the disk and the memory.

3.2 Self-Descriptive Representation of Structured Types

Given a structured type, a minimum value of the type (i.e., with all attributes set to null) consists of a fixed-size header, an attribute pointer array, and an attribute type array. The fixed-size header contains, among other things, the following fields:

- length: the total length of the structured typed value;
- type id of the structured type;
- the total number of attributes including inherited ones, which also indicates the size of the attribute pointer array and the attribute type array in the variable-length part of the structured type header;

It should be mentioned that the length field of a structured type value may not be exactly the number of bytes that it currently occupies in memory. A structured typed value may contain empty spaces inside when the new value of an attribute after mutation has a shorter length. If it is stored in a non-linearized form, the value of the length field may be less than the minimum length of the structured type in the linearized form due to the padding spaces required for alignment.

The variable-size part of a structured type header contains two arrays, one indicating the values of all the attributes and the other indicating the type of each attribute. The size of both arrays is the number of attributes in the structured type.

- attribute pointer array: each element points to the value of the corresponding attribute. A value zero indicates a null attribute value.
- attribute type array: each element indicates whether the corresponding attribute is of a base type, a structured type, or a large object type, and in case of a large object type, what kind of large object it is.

An attribute value of a base type or a large object type consists of a length field and a data field. If an attribute is of a structured type, the attribute value is represented in the same manner.

From an implementation perspective, when a user invokes an observer or a mutator method to access or mutate the attribute value of a structured typed value, the observer or mutator invocation is converted into an invocation of a builtin operation. Several builtin functions have been introduced for direct access and manipulations of structured types, including:

- three observer operations based upon the three categories of attributes: `adt_observe_base`, `adt_observe_lob` and `adt_observe_adt`. For base or large object attributes, the attribute value or a large object descriptor is copied into an output buffer. No output buffer is needed when observing an attribute of a structured type inside another structured type.
- three mutator functions based upon the three categories of attributes: `adt_mutate_base`, `adt_mutate_lob` and `adt_mutate_adt`. They mutate values of structured types *in place*. To implement the value-based semantics of mutator methods, there is also a builtin `adt_copy` function, which will be discussed in the next section.

Since values of structured types can be accessed and manipulated by users using the system generated observer and mutator methods and invocations of observer and mutator methods are converted into invocations of the builtin operations above, these builtin functions provide the only internal interface through which values of structured types can be accessed or mutated. Therefore any future changes in the representation of structured types can be localized to these builtin operations.

4 Copy Avoidance for Mutators

In SQL3 [6], mutator methods have no side effect and return values as results. However, our implementation uses builtin functions that update values of structured types *in-place*. A naive way to ensure the correctness with respect to the value-based semantics of mutator methods is to precede each mutator invocation with a copying operation so that mutation always takes place on a new copy. However, it leads to too many unnecessary copies of structured typed values. Consider the table `AddressBook` in Section 2.1 and the INSERT statement there:

```
INSERT INTO AddressBook
VALUES ('Mr.J.Smith',
        NEW ContactInfo()
        .postal_addr(
           NEW Address()
           .street('555 Bailey Ave')
           .city('San Jose')
           .state('CA')
           .zip(95141))
        .home_phone('6502347568')
        .work_phone('4084635678')
        .email('jsmith@us.ibm.com')
        .fax('4084631234'));
```

It has five mutator invocations for type `ContactInfo` and four mutator invocations for type `Address`, leading to five potential copies of `ContactInfo` values (including five copies of `Address` values inside) plus four potential copies of `Address` values. The excessive number of copies of structured typed values and the potentially large size of these values can cause significant performance overhead.

We have developed a simple compile-time algorithm that traverses a parse tree and determines when a copy operation is needed for a mutator method invocation.

When an SQL query is compiled, parse trees for invocations of the observer/mutator methods are transformed into parse trees for invocations of the builtin operations. Specifically,

- An invocation of the "NEW" notation is transformed into a call to the builtin function `adt_constructor` for constructing a new value of a structured type with all attributes set to null;
- A call to an observer method is transformed into a call to `adt_observe_base`, `adt_observe_lob` or `adt_observe_adt`, depending upon the type of the corresponding attribute;
- A call to a mutator method is transformed into a call to `adt_mutate_base`, `adt_mutate_lob` or `adt_mutate_adt`, depending upon the type of the corresponding attribute.

The copy avoidance algorithm for mutator methods is based upon the following observations:

- Expressions such as column references, which refer to values in a shared space by all transactions such as the buffer pool, cannot be mutated in-place. This also avoids the direct update of a table through mutator methods, which the SQL3 standard prohibits. Instead tables with columns of structured types can be updated only through the standard INSERT, DELETE and UPDATE statements.
- All other occurrences of expressions represent values that are used only once in an observer/mutator method invocation. Therefore if the incoming argument can be mutated safely in-place, the result can also be mutated safely in-place.
- Any call to a constructor results in a new value that can be mutated safely in-place.

The algorithm is implemented by associating, with each node in a parse tree, a `mutation_safe` flag that is initialized to `false`.

- If the parse tree node references a table column, set the flag to `false`;
- If the operation in the parse tree node is `adt_constructor`, set the flag to `true`;
- If the operation in the parse tree node is an observer (`adt_observe_{base,lob,adt}`), set the flag to the flag of the parse tree node for the subject of the observer method;
- If the operation in the parse tree node is a mutator (`adt_mutate_{base,lob,adt}`), then set the flag to `true` if the flag for the parse tree node of the subject of the mutator is true. Otherwise, insert an `adt_copy` function for the subject of the mutator and set the flag for the mutator operation to `true`.

5 Dealing with Variable Length Attributes and Subtyping

When values of a structured type are stored in the memory, they may not be in a linearized format in the sense that all attribute values may not follow after one another or follow immediately after the header. When they are stored in tables on the disk, they are linearized when they are written to the disk. Because of subtyping and variable length attributes such as large objects, values of a column of a structured type can have drastically different sizes. It may not be possible to store all of them inline since the size of the record buffer for a row in a table is often limited by the page size. On the other hand, it is not wise to store *all* of them out-of-line like large objects as accessing large objects in a separate storage space is more expensive. Our approach is to store values of structured types inline or out-of-line dynamically depending upon their actual sizes. In addition, large object attributes inside a structured type can also be stored inline whenever possible.

To provide some control by the user over the inline storage of structured typed values in a table column, we have introduced a new column option named INLINE LENGTH that can be specified for a structured typed column when a table is created. The value of INLINE LENGTH is an integer that represents a number of bytes. Column values whose size is larger than the inline length will be stored outside the table, in the form of a large object.

Inline Storage of Large Object Attributes

Structured types are often used to encapsulate large objects, e.g., to manage image and textual documents, together with other attributes and with methods. It is important to handle large object attributes inside structured types efficiently. While large objects often have a large maximum size, large object values in an application may have drastically different sizes. For example, a geometry large object may contain only a single point, a small polygon, or a large bitmap for an area. In general, large objects are represented by descriptors that indicate where the data are actually stored, usually in a separate tablespace. Therefore chasing large object descriptors to the actual data is an expensive operation.

To avoid these problems, We store *small* values of large object attributes of structured types in an *inline* fashion under two situations:

- If the size of a large object descriptor is larger than the size of the actual data, we always store its value *inline*.

- If the inline length specified for a table column can accommodate the inline representation of a large object attribute in a value of a structured type, then store the large object attribute *inline*. This decision is made based upon a "first-come first-serve" basis. For instance, if a value of a structured type contains two large object attribute values inside, the one that is encountered first may be stored inline, while the other large object value may have to be stored out-of-line and is represented by a large object descriptor.

Turning Values of Structured Types into Large Objects

Values of structured types are stored inline whenever possible. However, it is possible that even if every large object inside a value of a structured type is represented by a large object descriptor, the resulting size still exceeds the maximum size of the record buffer of a row. If that is the case, we turn the value of the structured type itself into a large object. The corresponding large object descriptor is then stored inline. Notice that there may still be large object values stored inline inside the value of the structured type when the size of the large object value is less than the size of a large object descriptor. This kind of "lobification" is applied to values of structured types that are not nested inside other values.

6 Performance Comparison

This section reports some preliminary performance results comparing several alternative implementations of structured types. All the queries are run using IBM DataJoiner 2.1.2 on a server machine.

We use a table for the census block data for the state of Kentucky with 137173 rows. The table has 10 columns of builtin types and one column of a structured type (called `polygon`). The type `polygon` has 13 attributes, one of which is a binary large object storing all points in the polygon.

```
CREATE TYPE Polygon
AS (srid int, numpoints int,
    geometry_type smallint,
    xmin double, ymin double,
    xmax double, ymax double,
    zmin double, zmax double,
    area double, length double,
    anno_text varchar(256),
    points blob(1m)) NOT FINAL;

CREATE TABLE census
(name varchar(20), rowid int,
 a1 decimal, a2 decimal, a3 decimal,
 a4 decimal, a5 decimal, a6 decimal,
 a7 decimal, a8 decimal, shape Polygon);
```

Several alternative implementations of structured types are considered:

- VARCHAR: The type of column `shape` is changed to `varchar(3930) for bit data`; Its values are truncated if necessary in this case. While the varchar representation has the advantage of inline storage, its size limitation is a serious shortcoming for representing values of structured types.

- BLOB: The type of column `shape` is changed to `blob(1m)`. Compared to VARCHAR, the BLOB representation has a much larger maximum size. However, the actual data for `shape` will be stored out-of-line, separate from the rest of the data in the table.

- SIDE TABLE: Column `shape` is eliminated and all values of the column are stored in a separate side table. Column `rowid` serves as the foreign key linking the side table to the original table containing other business data.

- FLAT TABLE: Column `shape` is expanded into multiple columns, one for each attribute of type `Polygon`. The difference from SIDE TABLE is that all attributes of the structured typed column `shape` are stored in the *same* table with other business data, avoiding the extra join in the SIDE TABLE representation.

Attributes of structured types are accessed by using builtin functions in DB2, user-defined functions in the VARCHAR or BLOB approach, or regular table column access in the SIDE TABLE or FLAT TABLE approach. User-defined functions can be either fenced or unfenced. If a user-defined function is fenced, it will be executed in a process or address space that is separate from that of the database manager. In general a function running as fenced will not perform as well as a similar one running as unfenced. Indexing is not used.

The following queries for the IBM DB2 approach and their corresponding variants for alternative implementations of structured types are executed:

- QALL — retrieve all attributes (except the annotation text) of column `shape` of the entire table (with 137173 records) :

```
SELECT shape.srid, shape.numpoints,
       shape.geometry_type,
       shape.xmin, shape.xmax,
       shape.ymin, shape.ymax,
       shape.zmin, shape.zmax,
       shape.area, shape.length,
       shape.points
FROM   census
```

- QSCALAR1 — retrieve some scalar attributes of column `shape` using a predicate on business data:

```
SELECT shape.numpoints,
       shape.xmin, shape.ymin,
       shape.area
FROM   census
WHERE  a1 = 0
```

  The query result contains 45417 records.

- QBLOB — retrieve the binary large object attribute of column `shape` using a predicate on column `rowid`:

```
SELECT shape.points
FROM   census
WHERE  rowid < 30001
```

  The query result contains 30000 records.

- QSCALAR2 — retrieve some scalar attributes of column `shape` using a predicate on column `rowid`:

```
SELECT shape.numpoints,
       shape.xmin, shape.ymin,
       shape.area
FROM   census
WHERE  rowid < 10001
```

  The query result contains 10000 records.

- QCOUNT1 — count the number of records using an equality predicate on an attribute of a structured type:

```
SELECT count(*)
FROM   census
WHERE  shape.numpoints = 100
```

- QCOUNT2 — count the number of records using a complex predicate on attributes of a structured type:

```
SELECT count(*)
FROM   census
WHERE  (shape.length
        between 1 and 200) and
       (shape.numpoints
        between 10 and 100)
```

For all queries except QSCALAR1 that has a dependency on the business data, there are two versions for the SIDE TABLE approach. One version involves column `rowid` in the main table and a join with the side table. The other version does not involve a join and, instead, retrieves data from the side table directly. The latter version assumes that the query optimizer is intelligent enough to avoid the join. This may not be trivial since users pose queries using the main table, unaware of the side tables being used to store values of structured types. The sole N/A entry in the table is for the case where the join with the side table cannot be avoided.

Table 1 shows the sum of the CPU time and synchronous I/O time (in seconds) for all the queries in different approaches.

For the first four queries, the access of attributes of structured types occurs in the SELECT clause. In the

Queries	Qall	Qscalar1	Qblob	Qscalar2	Qcount1	Qcount2
DB2	96.4	17.2	27	10.1	6.7	7.4
varchar (unfenced)	160	24.01	30.1	11.1	11.3	13
varchar (fenced)	320	33	35.5	16	29	60
blob (unfenced)	520	70	32.3	15.9	59	107
blob (fenced)	740	108	44.8	25.4	81	178
side table (w/join)	320.3	24.04	33.8	14.2	9.7	9.5
side table (w/o join)	159	N/A	29.7	10.1	4.3	4.1
flat table	182	18.1	29.56	11.8	5.6	6.7

Table 1: Performance measurements of retrieval of attributes of structured types

VARCHAR approach, even though all values of structured types are stored inline (and truncated in some cases), the overhead of invoking user-defined functions is much higher than that of executing builtin operations. The reason is that builtin operations are part of the database engine and do not require extra environment setup or protection to be run. When the number of such invocations increases (from QSCALAR2 to QALL), the relative performance of the VARCHAR approach to DB2 deteriorates. The use of fenced user-defined functions adds even more overhead. The BLOB approach is similar to VARCHAR except that all binary large objects are stored out-of-line and have to be retrieved and materialized for the invocation of user-defined functions.

The DB2 approach performs better than the side table approach for two reasons. First, the SIDE TABLE approach requires an extra join, which is normally the case since users access attributes of structured types in side tables from business tables and the side tables are hidden from users. Second, binary large object attributes inside a structured type can be stored inline whenever possible in the DB2 approach, while binary large objects inside a regular table column are stored out-of-line. When the extra join is eliminated *manually* in the SIDE TABLE approach and in the FLAT TABLE approach, DB2 still retains its performance advantage for queries QALL and QBLOB due to its dynamic inline/out-of-line storage of binary large object attributes in structured types and holds its ground for the access of scalar attributes in queries QSCALAR1 and QSCALAR2.

For the two count queries, the access of attributes of structured types occurs in the WHERE clause. The DB2 approach still performs better than the SIDE TABLE approach with join. However when the join is eliminated manually, the direct access of structured types in the side table (without going through the original table of business data) is more efficient. Similarly the FLAT TABLE approach also performs better in this case. The reason is that our attribute access is performed by builtin operations, but predicates involving observer/mutator methods are not pushed down deep enough into the database engine by the query optimizer.

7 Conclusion

We have presented an implementation of SQL3 structured types with inheritance in DB2. Its salient features include: (a) encapsulation of the implementation of structured types, minimizing the impact on low level components of the database engine; (b) compile-time copy avoidance algorithm for efficient implementation of the value-based semantics of mutator methods; and (c) dynamic inline/out-of-line storage of LOB attributes inside structured types and of structured types themselves. For future work, we intend to investigate further the performance and query optimization issues of structured types.

Acknowledgment

The authors thank Michael Carey for his detailed comments on an an early draft of this paper and thank the anonymous referees for their feedback on this paper.

References

[1] M. Carey, D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, and N.M. Mattos. O-O, what have they done to DB2? In *Intl. Conference on Very Large Data Bases*, September 1999.

[2] D. Chamberlin. *A Complete Guide to DB2 Universal Database.* Morgan Kaufmann Publishers, Inc., 1998.

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

[4] ESRI. Environmental System Research Institute (ESRI). Home page http://www.esri.com.

[5] Informix. Informix DataBlade Products, 1997. http://www.informix.com.

[6] ISO Final Draft International Standard (FDIS) Database Language SQL – Part 2: Foundation (SQL/Foundation), February 1999.

[7] M. Stonebraker and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave.* Morgan Kaufmann Publishers, Inc., 1999.