On the linkage of dynamic tables in relational DBMSs

by G. Y. Fuh
J.-H. Chow
N. M. Mattos
B. T. Tran
T. C. Truong

Tables and operations over tables are at the center of the relational model and have been at the core of the Structured Query Language (SQL) since its development in the 1970s. As database applications have grown rapidly, the concept of tables has been generalized in database languages. The new generalized table concept in the SQL standard and in some commercial databases includes explicitly defined derived tables, such as user-defined temporary tables, transition tables, user-defined table functions, and table locators, that can be manipulated by users. We call them dynamic tables, because their entities exist only at run time. The challenges that these dynamic tables pose to existing relational engines lie in the linkage between the creation of the derived table and its references. In this paper, we describe a uniform framework for compile-time and run-time processing of dynamic tables. We also give a thorough explanation of how such a generic framework can be realized in existing relational database management systems, such as IBM DATABASE 2™ Common Server. Our experience with our prototype has shown the simplicity, generality, and efficiency of our approach.

Tables and operations over tables are at the center of the relational model and have been at the core of the Structured Query Language (SQL) since its development in the 1970s. Queries define operations that accept tables as input operands and produce other tables as output. Query evaluation within a relational database management system (DBMS) engine is also based on relational operators (e.g., restriction, projection, join, etc.) that manipulate table record streams.

There are basically two types of tables supported by SQL: base tables and derived tables. In the SQL-92 standard,[1–3] base tables are used to store the data in the database. In contrast to base tables, derived tables are defined in terms of existing base tables or other derived tables. They can be defined either explicitly by the user or implicitly by the database engine. In SQL-92, explicitly defined derived tables, called "views," are specified by users in a CREATE VIEW statement. Implicitly defined derived tables are temporary tables created during the execution of table operations to store intermediate results, and in general they are not directly manipulable by the user. However, regardless of their type, tables are internally manipulated uniformly by the DBMS.

As database applications have grown rapidly, the concept of tables has been generalized in the SQL standard[4] and in some commercial DBMSs, for example, the user-defined temporary tables in SQL-92,[1] transition tables within triggers in SQL3,[5] and user-defined table functions in IBM DATABASE 2* (DB2*) Universal Database (UDB).[6]

These new derived tables are transient, but unlike implicitly defined derived tables, they are directly ma-

nipulable by the user. In all cases, the physical table produced by these extensions does not exist at compile time, nor is it created by the SQL statement that references it. For example, *transition tables* associated with a trigger are implicitly defined tables that will contain copies of the rows affected when the trigger is fired by some INSERT, UPDATE, or DELETE statement. The trigger body, defining the action when the trigger is fired, may reference these temporary tables. Because the trigger body is compiled before the execution of the SQL statement that fires the trigger, the actual transition table does not come to exist until the firing SQL statement is executed. So, how should a database engine represent these transition table references at compile time?

The challenge that these extensions pose to existing relational engines lies in the linkage between the creation of and the references to the derived table. This problem is similar to the problem of external name references that already exists in today's programming languages.[7] Analogous to the approach used by the programming language compiler, in the trigger example the SQL compiler will need to mark transition table references as *unresolved*. Resolution occurs when the trigger is fired and the trigger body is given the actual transition table for execution. In other words, the linkage between the table creation and the table reference takes place dynamically as opposed to statically. For this reason, we call such tables *dynamic tables*, to separate them from ordinary derived tables.

This process of linking a table reference to its physical table entity, called *dynamic linking*, is the subject of this paper. The contributions of our work are the following. First, we have found that the key to supporting various dynamic tables is dynamic linking, and we have devised a uniform framework for both compile-time and run-time processing of dynamic tables. The idea is to view dynamic tables as generalized functions that produce record sets. For each dynamic table, the compiler creates a table template, which contains information known at compile time, such as its column definition. Second, our approach has only small impact on existing run-time architectures. We isolate most changes to the run-time architecture into a new functional component, the *dynamic broker*, which is responsible for dynamically linking unresolved table references. Third, there is minimal performance impact. Dynamic linking takes place only when a dynamic table is first referenced (opened). Based on this approach, we have prototyped the support of several of these new table extensions (specifically table functions, triggers, table locators, and temporary tables) within the DB2 Common Server, which is an earlier release of the DB2 UDB. Our experience with this prototype has shown the simplicity, generality, and efficiency of our approach.

Since the use of dynamic tables is still relatively new, we briefly mention how transition tables for triggers are handled in some products. In DB2 UDB, the trigger body is compiled as part of an SQL statement that can fire the trigger; thus there is no dynamic linking issue. The disadvantage of this approach is the significant work required in the compiler to understand the semantics of triggers and, further, to prevent certain optimizations from being incorrectly applied. In DB2 for AS/400* (Application System/400),[8] the user creates a trigger program written in languages such as C or COBOL. When the trigger is fired, the trigger program is passed a pointer to a *trigger section* that contains information about the triggering statement and a buffer for the old and new records. Low-level application programming interfaces (APIs) are used to access the buffer records through code in the trigger program. None of these approaches is applicable to other dynamic tables.

The rest of this paper is organized as follows. The next section discusses dynamic tables and illustrates the issue of dynamic linkage. Following sections describe a compile-time framework, where dynamic tables are treated uniformly by the SQL compiler, and the extended run-time architecture for supporting dynamic tables. Remaining sections explain how the dynamic linkage process can be realized in the context of triggers, table locators, external table functions, and user-defined temporary tables, and conclude the paper.

Dynamic tables

This section introduces the dynamic tables that are of interest in this paper and illustrates the dynamic linking issue.

User-defined table functions. A *table function* is a function that returns a set of records. It not only provides a more general way (than a view) to compose new tables from existing tables, but also allows access to external data (e.g., data stored in flat files) using the same query mechanisms. For example, we may write the following DB2 UDB statement to define a table function avg_temp, implemented in C,

that results in a table of <city,date,temp> with the average daily temperature for a group of cities:

```
CREATE FUNCTION avg_temp ()
  RETURNS TABLE (city VARCHAR (30), date DATE,
                                    temp INTEGER)
  LANGUAGE C
  . . .
```

The keyword TABLE in the RETURNS clause indicates that the function is a table function.[9] Once defined, this function can be used in a query, for example, to return the average temperature in Chicago on July 13, 1959:

```
SELECT temp FROM TABLE (avg_temp ()) AS adt
  WHERE city = 'CHICAGO'
  AND date = DATE '1959-07-13'
```

Notice that the table avg_temp in the SELECT statement does not exist, nor is it accessible by the database, until the table function is executed at run time. In other words, the compiler has generated an executable *plan* (also called *access section*) for the SELECT statement that refers to a nonexistent table.

Transition tables in triggers. As mentioned briefly in the introduction, a transition table contains the set of rows that were affected by the triggering statement, i.e., those rows that are being inserted, updated, or deleted. The scope of a transition table is the whole trigger body, where it can be used as if it were a base or derived table.

The following defines a table employees and a trigger keep_stat that will be fired after updates on the table are performed:

```
CREATE TABLE employees
  (name     VARCHAR (30),
   salary   DECIMAL (9, 2),
   dept     VARCHAR (5))

CREATE TRIGGER keep_stat
  AFTER UPDATE ON employees
  REFERENCING NEW_TABLE AS new
  FOR EACH STATEMENT
  BEGIN ATOMIC
    INSERT INTO stat
      SELECT MIN (salary), AVG (salary), MAX (salary)
                                        FROM new
  END
```

The trigger body is defined by the statements within the BEGIN block. When fired, it inserts into the table stat a row with the new minimum, average, and maximum salary information from the set of affected rows with their *updated* values (REFERENCING OLD_TABLE could be used to refer to the affected rows with their *original* values). Given the trigger definition, the following UPDATE statement on employees, on execution, will create a transition table (specified as new in the trigger definition) containing the affected rows. In this case, the new table will contain the new records of employees in the sales department:

```
UPDATE employees SET salary = salary * 1.1
  WHERE dept = 'Sales'
```

The contents of the transition table are derived by the UPDATE operation and the trigger is fired after UPDATE is executed. Notice that the transition table referenced in the INSERT statement of the trigger body does not exist until the execution of the UPDATE statement, and again the compiler has to generate an executable plan for the INSERT statement that refers to a nonexistent table.

Table locators. Some SQL proposals suggest TABLE as a built-in data type. With such, *table locators* are introduced to bind tables (especially when they are used to define columns resulting from a query) to host variables.[10] Table locators are "handles" that allow applications to access the derived tables through regular SQL table operations within the same transaction. A host variable of a table locator type is declared in the DECLARE SECTION of the application program, as in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS TABLE (name     VARCHAR (30),
                     salary   DECIMAL (9, 2))
    AS LOCATOR emp_loc;
  SQL TYPE IS TABLE LIKE departments AS
                                LOCATOR dept_loc;
EXEC SQL END DECLARE SECTION;
```

One can declare a host variable of the table locator type by providing the complete table structure (i.e., the list of column names and data type pairs), or by providing the name of a table (departments in the above example) from which the table structure is to be derived. Once defined, the table locator host variable can be used in assignments or other SQL statements where tables can be used. In the following example, the host variable emp_loc is assigned the

result, of type TABLE, that contains names of the employees in the sales department who make more than $50 000:

```
EXEC SQL SET :emp_loc = (SELECT (SELECT *
    FROM TABLE (d.emps)
    WHERE salary > 50000)
  FROM departments AS d
  WHERE name = 'Sales');
```

Notice that the example statement does not really move the data of all employees of the sales department to the host program. It merely creates a derived table and assigns a handle value that uniquely identifies this derived table in the server, during the unit of work, to the host variable emp_loc. Because emp_loc uniquely identifies the derived table, subsequent queries can be issued, using this variable where SQL expects a table. For example, the following query returns the average salary of the employees in the table represented by emp_loc:

```
EXEC SQL SELECT AVG (salary) FROM TABLE
                                (:emp_loc);
```

User-defined temporary tables and other constructs. User-defined temporary tables are tables that are temporarily created and maintained by the SQL engine for application programs connected to the DBMS. They are defined like regular base tables, but do not contain any data until the execution time of a given application. The first time the application program references the temporary table, it is instantiated and made available for manipulation. In addition to temporary tables, SQL has other constructs that explicitly define derived tables for which the contents are not known until run time. These include named table expressions and result sets returned by stored procedures.

What is important to observe is that in all these constructs the structure of the explicitly defined derived tables is known by the SQL engine (since they are defined as regular tables), but the contents do not come into existence until the run time of an application program or SQL statement that references it.

Extended table object representation

In our introduction, we suggested a uniform way to view all kinds of dynamic tables: as functions that produce tables at run time. In this section, we describe how to represent these unresolved dynamic tables by extending the existing *table objects*, and in the next section, we describe how to use them for dynamic linking. For convenience, we assume an architecture similar to the DB2 Common Server.[11] We believe that the design presented in this paper applies also to other relational database systems.

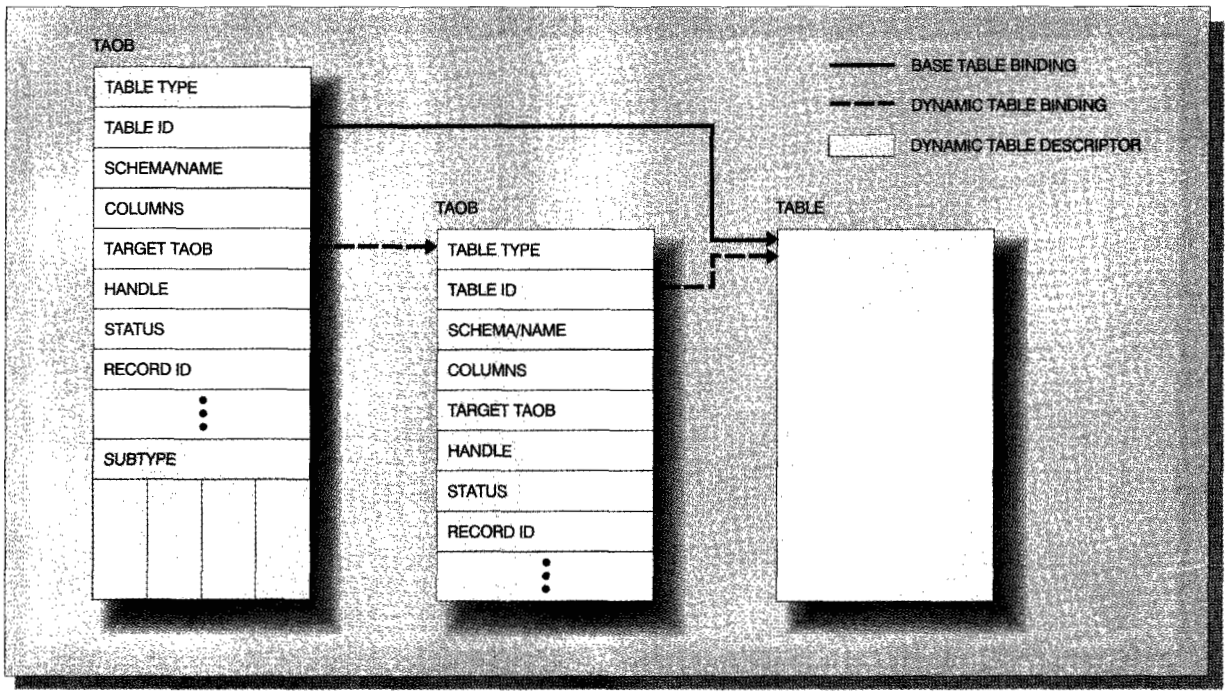
In DB2 Common Server, each SQL statement is compiled into an executable plan that consists of a set of run-time objects manipulated by threads of operators.[12,13] The main logic of a thread is to progressively construct intermediate tables by applying operators, such as *sort* or *join*, to the incoming table streams. The main data structure associated with any table operation is the table object (TAOB), and each table reference has its own TAOB.

Current table objects. A TAOB is a descriptor for a table reference in any table operation. Some of the TAOB attributes are known and set at compile time as constants (from the system catalog or the SQL statement context), for example, the table type and table identifier (ID) of a base table, the active column buffer areas, the associated search argument predicate,[14] etc. Some other TAOB attributes are used to keep track of the run-time state of the table, such as the current record ID, number of records fetched, status of last operation, etc. Figure 1 illustrates some important TAOB attributes that are of interest for this paper.

The type of a table is indicated by the first attribute. Currently, the possible values are *temporary* and *base*. The table ID uniquely identifies a table. For base tables, the table ID is known at compile time and is set by the compiler, while the table ID for a temporary table is set at run time, when the table is created. As shown in Figure 1, the compile-time TAOB has a pointer to a target TAOB. The table ID in the target TAOB points to the actual table. After the temporary table is created, all table operations on it will see the same table ID through their own TAOBs by this indirect pointer.

Many table operations are based on scans, either by relation or by index. To maintain the current state of a scan, the data manager component of the DBMS creates a handle structure at the time when the target table is opened. This handle structure keeps track of the position-sensitive information at the data access level, and the handle is used to access the target table for scan-based operations. The status field keeps the current status (e.g., open, closed, end of file) of the table operation. The record_id field keeps the ID of the last record fetched.

Figure 1 TAOB attributes



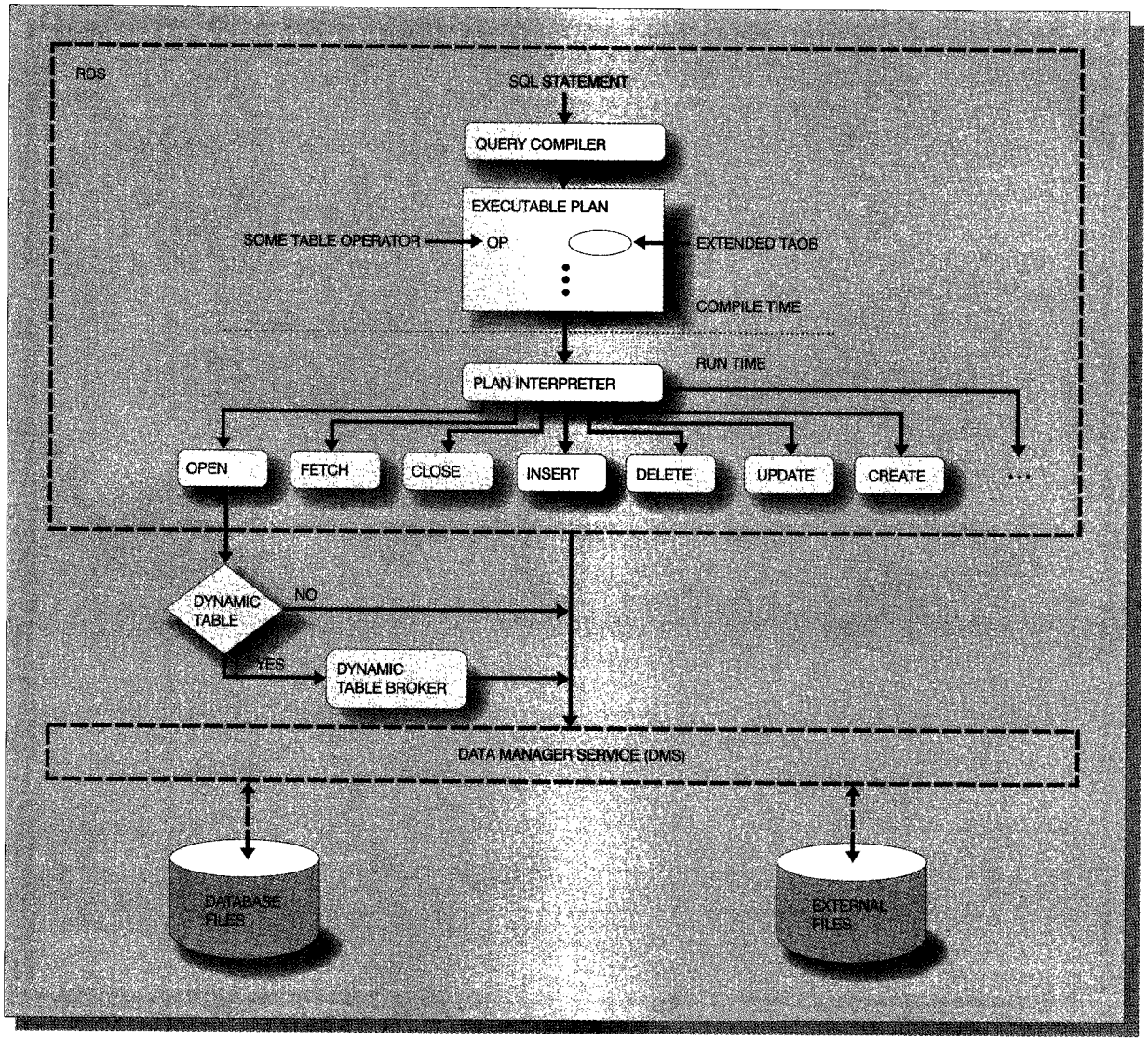
New attributes for dynamic tables. We now describe new fields in the TAOB for supporting dynamic tables. Unlike ordinary tables, a dynamic table needs to keep extra information obtained at compile time (from the SQL statement or the system catalog) in the TAOB, in order to resolve the "unknowns" at run time. The kind of information needed varies for the different dynamic tables:

- Table functions: The function invocation descriptor (UFOB) is needed for external table functions; the ID is needed for internal table functions.
- Transition tables: The type of transition table (old or new), the associated trigger name, and the associated base table name are needed.
- Table locator: The object that contains the column definition of the table locator is needed.
- User-defined temporary tables: The type of the temporary table (global, local, or declared local) and the associated table name are needed.

We have now identified these new table types: table function, transition table, table locator, and user-defined temporary table. A *dynamic table descriptor* is also added to the TAOB as the common control block for dynamic tables. For base tables and temporary tables, this is set to "null." The dynamic table descriptor contains the following information:

- Subtype: The subtype field is an extension of the table type. For table functions, it is set to internal or external. For transition tables, it is either new or old. For a user-defined temporary table, it can be global, local, or declared local.
- Table schema and table name: This is the schema and the table name with which the underlying dynamic table is associated. It applies only to transition tables and user-defined temporary tables.
- Trigger schema and trigger name: For a transition table, this attribute further describes the schema name and trigger name of the trigger where it is declared.
- Column definition object: This attribute points to the column definition of the corresponding table locator.
- External function object: This attribute points to the UFOB function descriptor of the corresponding external function invocation.

Figure 2 A typical relational DBMS architecture. A dynamic table broker is introduced for dynamic linking at the time the table is opened. Once table references are resolved, all table operations proceed as before.



In the following section, we will show how the extended TAOB described here is used at run time for dynamic linking between the actual table entities and the unresolved table references.

Extended run-time environment

A typical relational DBMS engine at run time consists of a relational data service (RDS) component for the logical (relational) view of the database, and a data manager service (DMS) component for the physical (raw data) view of the database. Figure 2 shows such an overall architecture. It also includes a new component, the *dynamic table broker*, that is used to support dynamic tables.

As indicated in this figure, access requests to base tables and ordinary derived tables will be processed uniformly as before. The interpreter invokes the corresponding relational data access routines, which will in turn invoke lower-level DMS data access routines. Since the TAOBs of the subject tables are already

"linked" to the table ID corresponding to the physical table entity for base tables at compile time, or at run time for ordinary derived tables, no special treatment is needed for them.

Access to dynamic tables is complicated by the fact that they are produced externally to the executing SQL statement, whereas ordinary derived tables are produced internally. The dynamic table broker component is introduced to carry out this dynamic linking process. In this section, we give a brief overview of the existing run-time environment and describe how the broker component can be integrated into the existing run-time routines.

Current run-time environment. Table entities, whether base or derived, must be created before they can be accessed. A base table entity must be created before any SQL statement that accesses it can be compiled. Its table ID is known *a priori* and is stored in the TAOB of operations that access the table. On the other hand, derived tables are "computed" from other tables in an executing SQL statement. Ordinary derived tables are created (and thus acquire a table ID), populated, and accessed when the table expression that (implicitly) defines the derived table is evaluated. These derived tables are normally dropped after execution of the SQL statement.

Any table must first be opened to initialize appropriate working areas before it can be manipulated. The open process is accomplished by invoking the *open table* run-time routine, which uses the table ID to invoke lower-level DMS routines. Once the table is opened, tuples in the table can be manipulated by each individual run-time routine. When table manipulation is completed, the *close table* run-time routine is invoked to release the working areas.

The dynamic table broker. Since all table operations have to go through the open routine, the natural place to do dynamic linking is at open time, so we add a simple dispatcher at the very beginning of the open table routine. If the type indicates a base table or a derived table operation, then the dispatcher allows the request to fall through to the existing logic. For dynamic tables, control will be passed to the dynamic table broker. The dynamic table broker will resolve the linkage between unresolved dynamic table references and the corresponding table entity. Once the linkage is resolved, all the run-time routines can proceed as if the underlying table were an ordinary derived table.

Dynamic linking involves two steps: first, finding the target table entity and, second, storing the obtained information in the current TAOB for subsequent uses. Although the details of table lookup and attribute setting vary from one dynamic table to another, the fundamental mechanism is the same. The following segment of C-like pseudocode demonstrates the dispatching logic of the dynamic table broker:

Dynamic linking has two steps: finding the target table entity and storing the obtained information in the current TAOB.

```
switch access_taob.type
{
  case TRANSITION_TABLE:
    link_transition_table(access_taob);
    break;
  case TABLE_FUNCTION:
    link_table_function(access_taob);
    break;
  case TABLE_LOCATOR:
    link_table_locator(access_taob);
    break;
  case USER_DEFINED_TEMPORARY_TABLE:
    link_user_defined_temporary_table(access_taob);
    break;
}
```

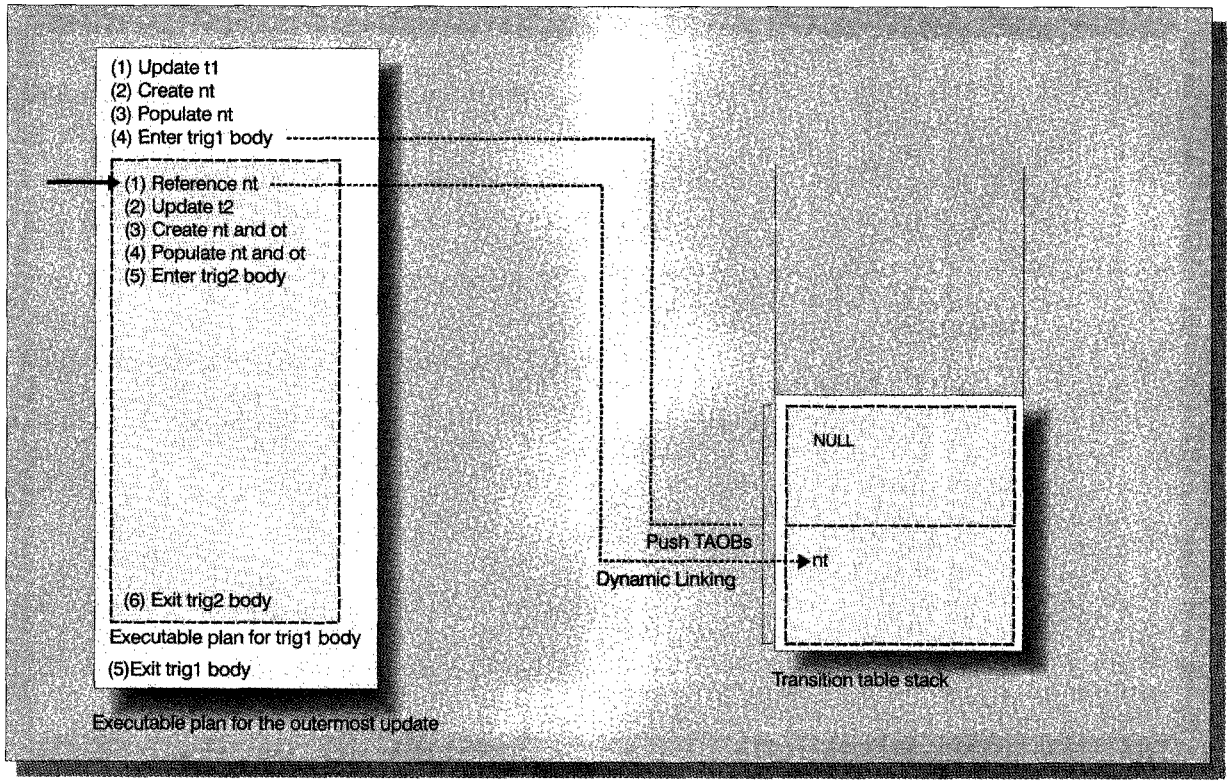
where link_transition_table(), link_table_function(), link_table_locator(), and link_user_defined_temporary_table() perform the linkage for the respective dynamic tables. The following section describes how these routines can be realized.

Supporting dynamic tables

This section describes how the extended run-time architecture supports dynamic tables.

Transition tables in triggers. Transition tables capture the state of affected rows when the triggering SQL operation is applied to a table. More specifically, the *old transition table* contains the value of affected rows prior to the application of an UPDATE or a

Figure 3 Run-time configuration and dynamic linking after trig1 is entered



DELETE operation, and the *new transition table* contains the value of affected rows that will be (or were) used in an UPDATE or an INSERT operation. When a triggering operation (INSERT, DELETE, or UPDATE) is executed, transition tables are created and populated, based on the subtype of the transition table (old or new), the triggering operation, and the current content of the table.

Two more details deserve further discussion. First, transition tables are created during the execution of the triggering statement. Therefore, the TAOB has to be recorded in a specific area known by the dynamic table broker routine, link_transition_table(), to resolve references to the transition table in the executable plan associated with the trigger body. Second, the activation of triggers can be nested, because some of the SQL statements in the trigger body may cause another (or the same) trigger to be activated. Therefore, like procedure calls in conventional programming languages, the data structure for maintaining TAOBs of transition tables is a stack so that the innermost TAOB always has precedence. TAOBs are pushed onto the *transition table stack* at trigger body entry and popped off at trigger body exit.

The dynamic table broker finds the top entry of the transition table stack for the actual table entity, and stores the obtained table ID in the TAOB of the current table reference. To illustrate our discussion, here is pseudocode for the main logic of the dynamic table broker routine link_transition_table():

```
link_transition_table(access_taob)
{
  TAOB actual_taob;

  /* (1) Look up the transition table stack */
  if (access_taob→dt_cb.subtype == NEW)
    actual_taob = tran_tbl_stack[top].new;
  else
    actual_taob = tran_tbl_stack[top].old;
  /* (2) Copy specific attributes */
  access_taob→type = actual_taob→type;
```

```
  access_taob->id    = actual_taob->id;

  /* (3) Copy implementation-dependent generic
                                          attributes */
  modify_other_attributes(access_taob, actual_taob);
}
```

Putting everything together, we use the following example to demonstrate the run-time flow of the dynamic linkage for transition tables. Let t1, t2, and t3 be tables with numeric columns c1 and c2, and trig1 and trig2 be the *after update for each statement* triggers for t1 and t2, respectively. Moreover, trig1 processes only the new transition table nt, whereas trig2 processes both the new and old transition tables nt and ot:

```
CREATE TRIGGER trig1 AFTER UPDATE ON t1
       REFERENCING NEW_TABLE AS nt
       FOR EACH STATEMENT MODE DB2SQL
BEGIN
    ...
  CASE (SELECT COUNT(*) FROM nt)
    WHEN 2:
      UPDATE t2 SET c2 = c2 * c2 WHERE c2 < 0;
  END CASE;
    ...
END

CREATE TRIGGER trig2 AFTER UPDATE ON t2
       REFERENCING   NEW_TABLE AS nt
                     OLD_TABLE AS ot
       FOR EACH STATEMENT MODE DB2SQL
BEGIN
    ...
  INSERT INTO t3 SELECT nt.c1, ot.c2 FROM nt, ot;
    ...
END
```

The following update operation on t1 will fire trig1, which will in turn fire trig2:

```
UPDATE t1 SET c2 = -c2 WHERE c2 < 0
```

The run-time configuration and the dynamic linking process when trig1 is fired are sketched in Figure 3. During the update of t1, the transition table nt is created and populated with affected rows. Before trig1 is entered, the TAOB of nt is pushed onto the transition table stack (the old transition table is not generated and is indicated by "null"), and then a dynamic linking occurs at the first reference to nt in the trigger body. Figure 4 shows the configuration when trig2 is also fired.

User-defined temporary tables. A *global temporary table* is a user-defined temporary table shared by all SQL operations in a database connection session. It is "global" in that changes are immediate and visible by subsequent operations against the same database connection. It is "temporary" in that the content persists only during the database connection; the physical table entity for the global temporary table is dropped at the end of the connection session. Another important characteristic of global temporary tables is that they are not shared among database connection sessions. These characteristics can be best understood through an example. Let gtt be a global temporary table and t1 and t2 be base tables with column definition identical to that of gtt. The database connections, serialized or interleaved, will populate t1 and t2 with the rows as shown in Figure 5.

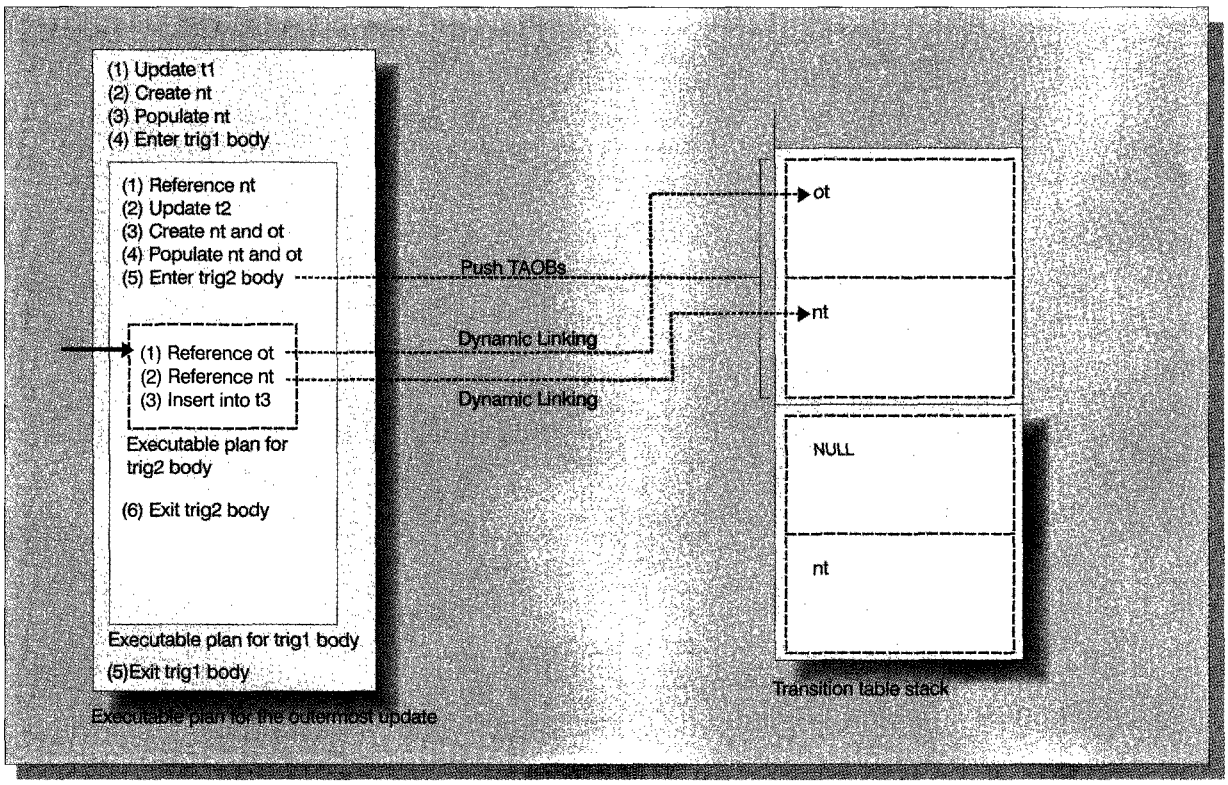
When a global temporary table is opened on behalf of an SQL operation, the dynamic table broker routine link_user_defined_temporary_table will look up the actual table entity in a global symbol table located in the working area for the connection. If it is found, the TAOB of the actual table entity is used to perform the dynamic table linkage. If it is not found, the table entity is created and entered into the symbol table for subsequent lookups. The TAOB of the newly created table entity is then used to perform the dynamic table linkage for the underlying table access. The details of the table linkage are identical to that of transition table reference, and the following pseudocode outlines the main logic of the dynamic table broker routine link_user_defined_temporary_table():

```
link_user_defined_temporary_table(access_taob)
{
  switch (access_taob->dt_cb.subtype)
  {
    case GLOBAL:
      {
        TAOB actual_taob;

        /* 1. Look up the global symbol table. */
        actual_taob = find_global_temporary_table
                                        (access_taob);

        /* 2. Create the global temporary table, if not
                                        created yet. */
        if (actual_taob == NULL)
```

Figure 4 Run-time configuration and dynamic linking after trig2 is entered




```
            actual_taob = create_global_temporary_table
                                              (access_taob);
            /* 3. Copy specific attributes. */
            access_taob→type = actual_taob→type;
            access_taob→id = actual_taob→id;

            /* 4. Copy implementation-dependent generic
                                              attributes. */
            modify_other_attributes(access_taob,
                                              actual_taob);
        }
        break;
    case LOCAL:
        {
          . . .
        }
  }
}
```

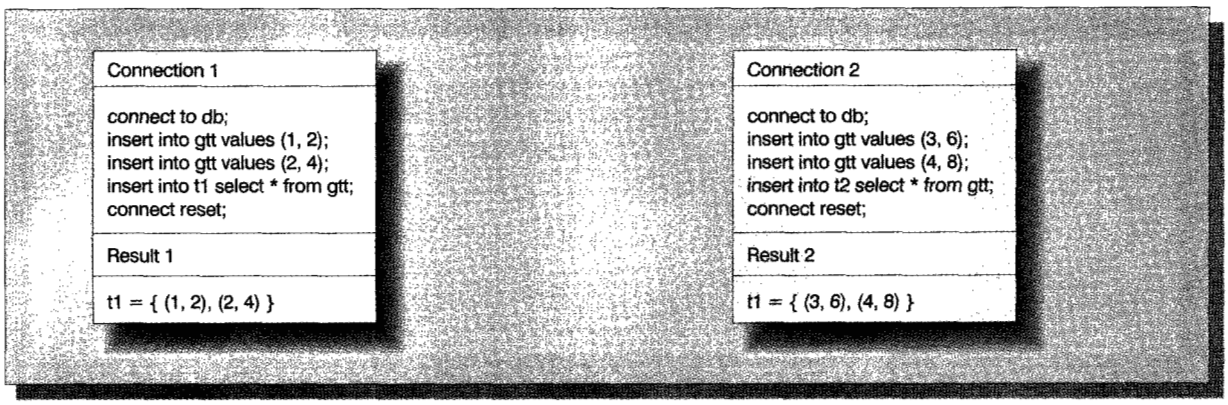
At the end of a database connection, the database engine will scan the global symbol table and drop all the table entities created on behalf of global temporary table accesses in that connection session.

Local temporary tables and *declared local temporary tables* are similar to global temporary tables. The only difference is in scope. Local temporary tables are shared among all the SQL operations belonging to the same SQL module, while declared local temporary tables are shared in a PSM (Persistent Stored Module[4]) basic block. Therefore, the data structure and table resolution logic for global temporary tables also apply for local and declared local temporary tables. However, since the symbol table has to be scoped, the initialization and the clean-up logic described above for global temporary tables has to be performed for each scope of the underlying SQL statement.

Table functions. Table functions can be internal or external. The body of internal table functions consists of a sequence of SQL statements. In contrast, external table functions are written in host languages such as C, C++, Java**, Visual Basic**, etc.

For internal table functions, the physical table entity is indeed the executable plan of the body of

Figure 5 Global temporary tables



the table function. Therefore, the broker routine link_table_function invokes the plan manager of the database engine to load the desired plan, which is identified by the plan ID stored in the dynamic table descriptor of the underlying TAOB. The in-memory descriptor of the plan being loaded is also recorded in the TAOB for execution of subsequent invocations of the table function.

External table functions are treated by the database engine as "black-box table producers." Therefore, the physical table entity is the entry point of the external function that implements the table function. To obtain the entry point of the external function, link_table_function dynamically loads the desired function library into the address space of the database engine. The symbol table of the library being loaded into memory is then searched for the desired external function. Once resolved, the function entry point is recorded in the dynamic table descriptor of the underlying TAOB for subsequent invocation of this table function.

The main logic of link_table_function is illustrated by the following pseudocode:

```
link_table_function(access_taob)
{
  TAOB actual_taob;
  switch (access_taob→dt_cb.subtype)

    case INTERNAL_TF:
        /* Internal table function: load access plan. */
        access_taob→dt_cb.plan_cb_ptr =
          load_plan(access_taob→dt_cb.planID);
        break;

    case EXTERNAL_TF:
        /* External table function: load library and
                                        resolve entry. */
        library_handle = dynamicLoad
                    (access_taob→dt_cb.libPath);
        access_taob→dt_cb.function_entry =
          resolveEntry(library_handle,
            access_taob→dt_cb.functionName);
        break;
}
```

Currently, the table function support in DB2 UDB allows only read operations (i.e., open, fetch, and close).

Table locators. A locator table is used to keep track of table locators created in a given transaction. The locator table maps a locator ID to a pointer to the TAOB of the actual associated table. At the beginning of a transaction, the locator table is initialized with no entry in it. During a transaction, new table entries are created whenever (derived) tables are "bound out" to a host variable (through the SET statement, for example). At the end of a transaction, the locator table is purged after all the table locators are freed.

When a table locator is accessed in an SQL statement, the run-time dynamic linking process takes place in two steps. First, the locator ID, which has been stored in the host variable, is bound in and stored in the dynamic table descriptor of the underlying access TAOB, at the bind-in time of the plan execution of the SQL statement. In the following pseudocode,

userObject is the locator ID from the host variable, and sqlObject is the access TAOB:

```
bind_in(type, sqlObject, userObject)
{
  switch (type)
  {
    case TABLE_LOCATOR:
      sqlObject→dt_cb.locatorID = userObject;
      break;
    case . . .
  }
}
```

Second, at the time the corresponding table is opened, the locator ID set earlier in the access TAOB is used to look up the locator table to get the actual TAOB. Then the TAOB attributes of the current TAOB are initialized according to those of the actual TAOB:

```
link_table_locator(access_taob)
{
  TAOB actual_taob;

  /* 1. Look up the global symbol table. */
  actual_taob = find_table_locator
                    (access_taob→dt_cb.locatorID);

  /* 2. Copy specific attributes. */
  access_taob→type = actual_taob→type;
  access_taob→id = actual_taob→id;

  /* 3. Copy implementation-dependent generic
                                        attributes. */
  modify_other_attributes(access_taob, actual_taob);
}
```

Conclusion

The traditional concept of tables in relational databases has been generalized in the SQL language and in some commercial database systems. Unlike traditional tables, dynamic tables exist only at query-execution time and are directly manipulable by the user. The query compiler generates unresolved table references and relies on some run-time linking mechanism to resolve them.

We have proposed a generic framework for supporting dynamic tables in existing query compilers, where all dynamic tables are treated in a uniform way by the compiler and broker functions are added into the run-time environment to establish the dynamic linkage. We have described the extensions to the compiler and the run-time environment in the context of DB2 Common Server, and explained how this generic framework can be applied to support transition tables, table functions, user-defined temporary tables, and table locators. We have also built a prototype based on this framework. The success of our prototype has confirmed our expectation of the simplicity and the applicability of our design.

In future work, we would like to continue exploring in two directions:

- Abstract tables: Recently, *abstract tables* have been proposed.[15] These would ultimately allow all operations on a table to be user-definable; that is, the user could define open, fetch, close, insert, update, delete, and even rollback and commit operations. Although we do not expect any impact from such a generalization on the framework we describe in this paper, we would like to take a closer look at the language specification.
- MPP parallel environment: DB2 UDB Version 5 supports MPP (massively parallel processing) where tables can be partitioned across multiple nodes to exploit a parallel environment.[6] A copy of the same executable plan is executed on multiple nodes, with table queues for shipping data streams. The resulting table from each node is piped through table queues to the "coordinator" node for final processing and bind out to the client. We would like to enhance our framework to work in such a parallel environment.

Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions on improving the presentation of the paper.

Cited references and notes

1. ISO-ANSI, *Database Language SQL*, ISO/IEC 9075: 1992, American National Standards Institute, 11 West 42nd Street, New York, NY 10036 (1992).
2. C. J. Date and H. Darwen, *A Guide to the SQL Standard*, Third Edition, Addison-Wesley Publishing Co., Reading, MA (1993).
3. J. Melton and A. R. Simon, *Understanding the New SQL: A Complete Guide*, Morgan Kaufmann Publishers, San Francisco, CA (1993).
4. ISO-ANSI, *Working Draft Database Language SQL/Founda-*

tion (SQL3), X3H2-97-315, DBL: BBN-008, J. Melton, Editor (September 1997); American National Standards Institute, 11 West 42nd Street, New York, NY 10036.
5. In this paper, we use SQL3 to refer to the next release of the SQL standard, which will be informally called SQL-98 or SQL-99, depending on when it is published, and we use SQL4 to refer to the release after SQL3.
6. *IBM DB2 Universal Database SQL Reference Version 5*, S10J-8165-00, IBM Corporation (1997); available through IBM branch offices. Also available from http://www.software.ibm.com/cgi-bin/db2www/library/pubs.d2w/report#UDB PUBS.
7. *IBM VisualAge for C++ Reference Manual*, S33H-4982-00, IBM Corporation (1997), available through IBM branch offices.
8. *IBM DB2 for AS/400 Database Programming*, SC41-5701-00, IBM Corporation (1997); available through IBM branch offices. Also available from http://booksrvr.rchland.ibm.com/bookmgr/8.htm.
9. Currently, TABLE is not a data type. The proposal of TABLE as a data type has been moved to SQL4.
10. A host variable is a variable in the (host) application's programming language.
11. *IBM DB2 SQL Reference for Common Server*, S20H-4665-01, IBM Corporation (1993); available through IBM branch offices.
12. G. M. Lohman, "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," *Proceedings of the ACM SIGMOD Conference*, Chicago, IL (June 1988), pp. 18–27.
13. L. M. Hass, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst," *Proceedings of the ACM SIGMOD Conference*, Portland, OR (May 1989), pp. 377–388.
14. P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference*, Boston, MA (May 1979), pp. 23–34.
15. ISO/ANSI, *Working Draft Management of External Data (SQL/MED)*, X3H2-97-321/DBL: BBN-014, J. Melton, Editor (October 1997); American National Standards Institute, 11 West 42nd Street, New York, NY 10036.

Accepted for publication May 12, 1998.

Gene Y. Fuh *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: fuh@us.ibm.com).* Dr. Fuh received the Ph.D. degree in computer science from the State University of New York at Stony Brook in 1989. Since then, he has worked in the area of compiler development for various programming languages, such as VHDL (Very High Scale IC [hardware] Description Language), Verilog, FORTRAN 90, and SQL. He is currently the technical leader for the DB2 Spatial Extender team and an architect for the DB2 UDB object-relational technologies. Prior to joining IBM in 1993, Dr. Fuh held several technical management positions in the electronic CAD (computer-aided design) industry. His recent technical interests are compiler construction, language design, object relational DB technologies, client/server debugging methodology, and Internet application development.

Jyh-Herng Chow *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: chowjh@us.ibm.com).* Dr. Chow received the M.S. and Ph.D. degrees from the University of Illinois at Urbana-Champaign in 1990 and 1993, and the B.S. degree from National Taiwan University in 1985, all in computer science. He has been working on the DB2 Universal Database engine and was a technical leader in developing shared-memory intraquery parallelism support in Version 5. His recent work focuses on Internet applications and the management of structured documents, such as those written in XML (eXtensible Markup Language). Prior to joining IBM's Database Technology Institute (DBTI), he was with the Application Development Technology Institute, responsible for developing shared-memory parallelizing compilers, run-time systems, and advanced compiler optimization techniques.

Nelson M. Mattos *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: mattos@us.ibm.com).* Dr. Mattos is a Senior Technical Staff Member and manager of DBTI. He is IBM's chief architect for object-relational DBMSs and the standard project authority for SQL. He leads extensions to SQL, drives the development of object-relational extensions for the DB2 products, and is a key force behind the various Extender products that exploit the object-relational features of DB2. His DBTI organization is working on several extensions to DB2 UDB: object-relational constructs, event management, support for multimedia, query rewrite and optimization, query parallelism, new index management techniques, component-based object and application development, and constructs to increase the expressive power of SQL. He has also been heavily involved in the development of the SQL3 standard as IBM's representative to the American National Standards Institute (ANSI) SQL committee and a U.S. representative to the International Organization for Standardization (ISO) Committee for database. He has contributed extensively to the design of SQL3 through more than 300 accepted proposals. Prior to joining IBM, Dr. Mattos was an associate professor at the University of Kaiserslautern, Germany, where he was involved in research on object-oriented and knowledge base management systems. He received the B.Sc. and M.Sc. degrees from the Federal University of Rio Grande do Sul, Brazil, in 1981 and 1984, and the Ph.D. degree in computer science from the University of Kaiserslautern in 1989. He has published over 30 papers on object-relational databases, knowledge base management, and application areas in various magazines and conferences, and is the author of *An Approach to Knowledge Base Management*, Springer-Verlag (1989).

Brian T. Tran *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: bttran@us.ibm.com).* Mr. Tran received the M.S. degree in computer science engineering from San Jose State University, California, in 1986. He is currently a key technical member in the Object Strike Force at DBTI. He is also a key developer in the Data Joiner/Spatial Extender, working to extend the current DB2/Data Joiner component to support spatial queries.

Tuong C. Truong *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: tctruong@us.ibm.com).* Mr. Truong received the B.S. degree in computer science in 1990 and the M.B.A. degree in 1998, both from San Jose State University. He joined IBM at the Santa Teresa Laboratory in 1991 and has worked at DBTI in the DB2 UDB SQL compiler area for the past five years. Mr. Truong's interests include database technology and Web technology and their business applications.

Reprint Order No. G321-5689.