Implementing Embedded Valid Time Query Languages

Abstract: Application development on top of database systems is heavily based on the existence of embedded and 4GL languages. However, the issue of designing and implementing embedded or 4GL temporal languages has not been addressed insofar. In this paper, we present a design approach for implementing an embedded temporal language that supports valid time. Furthermore, we introduce implementation techniques that can be used for implementing any embedded temporal language that supports valid time on top of a DBMS.

1. Introduction

In the past years temporal databases have received substantial research attention, and numerous application areas which would benefit from the introduction of the time dimension have been identified (e.g. [2], [5], [6], [7]). However, little has been reported on the implementation of temporal applications using the temporal database management systems (TDBMSs) implemented, with the exception of time series-oriented applications: in [8] an application for managing medical temporal data is presented, while [16] reports on a temporal application which was developed for Clinica Puerto de Hierro of Spain. One of the main reasons for this shortage, is the absence of suitable development tools: database application development is based on embedded and 4GL languages (usually complemented by graphical user interface libraries), and none of the temporal database systems implemented insofar ([3]) is reported to provide such tools. Some of these systems provide graphical user interfaces ([4], [8], [19]), but these interfaces are oriented towards assisting query formulation, and not targeted for general-purpose application development. TSQL2 ([22]) includes specifications for cursors, but no implementations for TSQL2 cursors has been reported insofar. Time series applications, on the other hand, can be easily developed since there do exist database systems whose development packages support the time series concept ([15], [17]). This paper presents a design approach to the implementation of an embedded temporal language for a TDBMS supporting valid time (VTDBMS). We will not focus on a specific temporal language but,

rather, we will identify *categories of statements* that appear in VTDBMSs, and outline algorithms for handling such statements. The design is based on the following assumptions:

- The non-temporal part of the valid time query language (denoted as VTQL, hereafter) is pure SQL, or can be mapped to it. For simplicity, in our discussion we will assume that the non-temporal part is pure SQL.
- A *terminal monitor* application that accepts and processes interactive VTQL statements has been developed. The terminal monitor uses certain algorithms to evaluate queries with temporal aspects. The design of embedded VTQL (denoted as VTQL, hereafter) does not rely on the existence of *specific* algorithms, but requires that such algorithms *do exist*.
- EVTQL will be implemented on top of a DBMS that supports embedded SQL. This implies that a *layered architecture* is adopted, as suggested in [20], [21] and [23].

Finally, for simplicity in our discussion, we will assume that *homogeneous valid time* and *tuple timestamping* ([13]) is used, although the techniques presented here may easily be adapted to handle heterogeneous valid time and attribute timestamping.

The remnant of this paper is organised as follows: section 2 outlines the syntax and semantics of EVTQL, and the categories of statements that may appear in a VTDBMS are identified. Section 3 presents a design approach to the implementation of EVTQL, discussing the software modules that compose it, the role of each module and the algorithms it employs. Although the design has been completed for both static and dynamic EVTQL statements, only static statements are presented in this paper, for brevity reasons. Section 4 concludes and outlines future work.

2. An Embedded Valid Time Query Language

EVTQL should obey the *dual-mode principle* ([10]), i.e. every VTQL statement that can be used interactively, may be executed as part of an application program. Embedded VTQL statements, however, may use a number of extensions not available to interactive statements (the prefix `EXEC VTQL` that is used to distinguish embedded VTQL statements from host language statements is a trivial issue, and will not be addressed in the rest of this document):

1. embedded VTQL statements may contain references to host language variables, which must be prefixed with a colon (`:`). Variable references may appear in any place of a VTQL DML statement that a literal can be used, and in the `INTO` clause.
2. the embedded `SELECT` statement is augmented with an `INTO` clause, which specifies the host variables into which data will be retrieved.
3. a number of statements (or *statement forms*) associated with *cursors* is available (`DECLARE CURSOR`, `OPEN`, `FETCH`, etc.), providing a *row-at-a-time* interface to the database.

In the remainder of section 2 EVTQL DML statements will be presented. DDL statements are not discussed, since their syntax and semantics is identical to their interactive VTQL counterparts. The `CONNECT`, `DISCONNECT` and `BEGIN/END DECLARE SECTION` statements are also excluded from the discussion, since they are identical to their embedded SQL counterparts ([9]). Each of the `SELECT`, `INSERT`, `DELETE` and `UPDATE` statements are subdivided into two categories, which will be used in the discussion in section 3:

- *Snapshot-transformable statements*. This category includes VTQL statements that can be mapped to a *single SQL statement*. Statements that include operations on data types representing time may fall in this category, even if the underlying DBMS does not directly support these operations: indeed, for a DBMS not supporting the TSQL2 `PRECEDES` predicate, the expression `instant1 PRECEDES instant2` may be rewritten as `instant1 < instant2` or even `precedes(instant1, instant2)`, if the underlying DBMS implements the *precedes* predicate as an operator or a function, respectively.
- *Non snapshot-transformable statements*. This category includes VTQL statements that require the application of a set operation not directly supported by the underlying DBMS. Such operators are coalescing operators ([13]), operators converting between instant and period timestamping, as well as the SL and SP_p operators introduced in [22] (chapter 27). If data are stored in the base tables in coalesced form, `INSERT`, `DELETE` and `UPDATE` statements against valid time tables always fall in this category.

2.1 Operations not involving cursors

The DML statements of VTQL that do not involve cursors are the "singleton" form of `SELECT`, the `INSERT` statement, and the `DELETE` and `UPDATE` statements, except for their the `WHERE CURRENT OF` forms, which will be presented later. These statements are described in the following paragraphs.

2.1.1 The SELECT Statement ("singleton" form)

The "singleton" form of the embedded VTQL `SELECT` statement is used to fetch at most one tuple from the database. Its syntax is identical to the interactive counterpart, except that the `INTO` clause may precede the `FROM` clause, specifying the host language variables into which the query results will be stored. For example the TSQL2 ([22]) query

```
SELECT Name
INTO :emp_name
FROM Employee E1
WHERE VALID(E1) CONTAINS PERIOD(DATE '01/01/1991', DATE '12/31/1991');
```

can be used to fetch the name of the employee that has worked during all of 1991 (assuming that only one such employee exists).

2.1.2 The INSERT Statement

The syntax of the EVTQL `INSERT` statement is identical to the corresponding statement of SQL, with two exceptions:

1. If the values to be inserted are specified via a *select*-query, the query may include temporal features.
2. If the column storing the tuple's valid time is *implicit* (as in TSQL2), a specific clause must be introduced to specify the tuple's valid time, when inserting data into a valid time clause using the `values` clause.

2.1.3 The DELETE and Update Statements (non-Cursor Forms)

The `DELETE` and `UPDATE` statements of embedded VTQL are used to delete and update either whole tuples, or specific periods of the valid time data stored in a table. The syntax of the embedded VTQL `DELETE` and `UPDATE` statements is identical to their SQL counterparts, with the following exceptions:

1. The `WHERE` clause in both statements may use temporal extensions.

2. When deleting or updating valid time tables, the valid time period to which the operation applies must be specified. If the column storing the tuple's valid time is implicit, special provision for modifying the tuple's valid time must also be included.

2.2 Operations involving cursors

Cursors provide a *row-at-a-time* interface to the database. Using cursors, an application may obtain addressability to tuples stored in the database (one tuple at a time), fetch data values into its address space, as well as delete or modify the tuples.

Before any cursor is used, it must be *declared*, i.e. associated with a VTQL `SELECT` statement. The syntax for cursor declaration is

```
EXEC VTQL DECLARE cursor_name CURSOR FOR VTQL-Select-Statement;
```

Cursor declaration does not trigger the evaluation of the associated query. The query is evaluated only when the cursor is *opened*, which is accomplished via the statement

```
EXEC VTQL OPEN cursor_name;
```

Upon opening, a cursor does not point to any tuple in the result set. In order to position the cursor to the next tuple (marking it as the *current tuple* of the cursor) and fetch the values of the selected attributes into host language variables, the statement

```
EXEC VTQL FETCH cursor_name INTO host-variable-list;
```

must be used. If no more data is available in the result set to be fetched, the *vtqlca.vtqlcode* variable (which is implicitly declared in all programs using embedded VTQL) is assigned a special value to indicate the error. Finally, a cursor may be closed using the statement

```
EXEC VTQL CLOSE cursor_name;
```

Two issues are worth noting here, with respect to TSQL2:

1. TSQL2 introduces *nested cursors* for valid time tables. *Outer-level* cursors provide addressability to *explicit attributes*; the *inner level* cursors provide access to the valid timestamps of the tuple, while the outer level cursor points to a specific tuple. In a relational environment, this implies that valid timestamps are stored in a different table, necessitating thus the need for joins in query processing. Since such a scheme is inefficient to implement in a relational environment, we will not consider nested cursors; instead, standard single-level cursors provide access *both* to the explicit attributes *and* the valid time of the tuple.

2. The cursor versions of the DELETE and UPDATE statements in TSQL2 apply only to *complete tuples*; the design presented in this paper provides with means to specify that the operation applies only to *a specific part* of the valid time of the tuple that is pointed to by the cursor.

2.2.1 The DELETE Statement (Cursor Version)

The syntax of the cursor version of the DELETE statement is

```
DELETE FROM table_name WHERE CURRENT OF cursor_name [VT_selection];
```

where *VT_selection* is a syntactic construct (probably a clause, such as the VALID clause in the non-cursor form of TSQL2's DELETE statement), specifying the valid time period to which the deletion applies. There is no particular reason why this syntactic construct should be placed last in the query: it may be positioned in any place, provided it does not introduce syntactic ambiguities.

If *VT_selection* is used, the cursor does not point to a *current tuple* after the UPDATE statement is executed, and thus is not qualified for further deletions and updates, until a new FETCH statement is executed for that cursor. If this specification is omitted, the deletion applies to the whole tuple that is current for the designated cursor.

The cursor version of the DELETE statement may be used only if the cursor points to an *updatable result tuple set*. The cases for which a result tuple set is updatable practically depend on the underlying DBMS ([12], [14], [18]). The SQL standard ([9]) specifies a minimal set of queries yielding updatable results (roughly, this includes queries which apply only SELECT and PROJECT relational algebra operations, with no duplicates removal, on a single base table; joins are permitted but only when expressed via a subquery), but various RDBMSs allow updates on the results of a wider range of queries. We will consider updatable only the result sets which are designated as updatable by the SQL standard. The usage of the coalescing operators, operators converting between period and instant timestamping as well as the SL and SP_p operators destroys the physical one-to-one mapping of result tuples to stored tuples (i.e. a tuple in the result set does not correspond to a single stored tuple), and thus renders the result set not updatable.

2.2.2 The UPDATE Statement (Cursor Version)

The syntax of the cursor version of the `UPDATE` statement is

```
UPDATE table_name SET col = value {, col = value}
WHERE CURRENT OF cursor_name [VT_selection];
```

When using this form of the `UPDATE` statement (which is permitted only if the cursor points to an updatable result tuple set), the update applies to the tuple that is current for cursor *cursor_name*. Again, *VT_selection* specifies the valid time period to which the operation applies. If *VT_selection* is used, the cursor does not point to a *current tuple* after the `UPDATE` statement is executed, and thus is not qualified for further deletions and updates, until a new `FETCH` statement is executed for that cursor.

3. Design of an EVTQL Architecture

In this section we describe the design of an architecture for the embedded VTQL language, which may be implemented on top of any relational DBMS. The proposed architecture follows the approach taken by embedded SQL packages, but introduces two additional modules, the *EVTQL preprocessor* and the *EVTQL library*. The application programmer develops the temporal application in some host language (e.g. C, C++ etc.), using embedded VTQL statements to interact with the database. Each file written by the programmer is then processed by the EVTQL preprocessor, which translates EVTQL statements to calls to the EVTQL library. Calls to the EVTQL library may need to be supported by data structures and/or embedded SQL statements, thus the output of the EVTQL preprocessor is an intermediate file conferment to the embedded SQL syntax rules. These intermediate files are fed to the DBMS's embedded SQL preprocessor, whose output is processed by the host language compiler, producing object files. Finally, object files are linked with the host language's libraries, the DBMS libraries, the EVTQL library and any other libraries specified by the programmer (e.g. GUI libraries), in order to produce the application executable file.

The EVTQL library is a collection of procedures, providing support for the execution of EVTQL statements and for temporal cursors. It is developed in some host language, and uses embedded SQL to interact with the database. The files containing the EVTQL library's source code are processed by the DBMS's embedded SQL preprocessor and are then compiled to

object files, using the host language compiler. Finally, the system library manager is used to bundle these object files into a single library file. Only this library file is needed in the linking phase of temporal applications.

Figure 1 illustrates the overall process of temporal application development, using embedded VTQL.

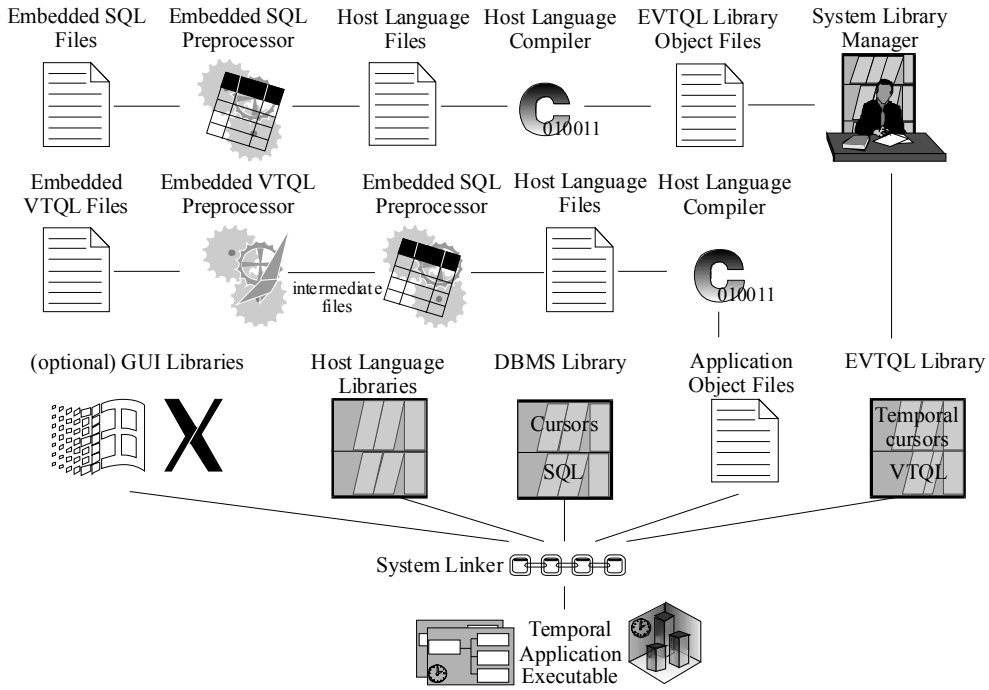

Figure 1 - Embedded VTQL architecture

In the following paragraphs, the operation of the EVTQL preprocessor and the algorithms employed by the EVTQL library are presented.

3.1 The EVTQL Preprocessor

The EVTQL preprocessor operates on files containing host language statements and embedded VTQL statements. For each input file, the embedded VTQL preprocessor generates an output file, which contains host language, embedded SQL statements and calls to the EVTQL library, and is then processed by the DBMS's embedded SQL preprocessor and the host language compiler, to produce object files. More specifically, the embedded VTQL preprocessor is responsible for the following actions:

1. analyse syntactically the embedded VTQL statements, verifying that they conform to the EVTQL syntax rules.

2. determine the types of the host language variables which are used within embedded VTQL statements, so as to facilitate information exchange between the VTDBMS and the application program.
3. translate embedded VTQL statements to appropriate calls to the EVTQL library. Each call must be supplied with parameters, which provide adequate information for the statement execution. Some statements need additional data structures and/or code to be introduced; the EVTQL preprocessor is responsible for planting these data structures and the appropriate pieces of code into the output file.
4. insert pieces of code to implement the `WHENEVER` statements[1].

In order to perform these tasks, the EVTQL preprocessor must be aware of the host language syntax and semantics rules, so as to tailor its behaviour to the specific host language (type analysis of variable declarations, code emission etc). In this paper, we focus on the EVTQL preprocessor for the "C" language, since C (along with C++) is the dominant third generation language for application development. The techniques presented, however, can be used for implementing preprocessors for other host languages. Since statement parsing is covered in detail elsewhere (e.g. [1], [11]), we focus only on the VTQL-specific aspects of the preprocessor, i.e. the translation of embedded VTQL statements and the implementation of the `WHENEVER` statements.

3.1.1 Embedded VTQL statement translation

A. The SELECT statement ("singleton" form)

The EVTQL preprocessor translates each singleton `SELECT` statement to a call to the EVTQL library function `evtql_select`. The function is supplied with two arguments, the second one being a description of the variables listed in the `INTO` clause. For each variable, its type, its length in bytes and a pointer to the memory location where the variable is stored are included in the description. If the `INTO` clause is omitted, a special value is used for the second argument. Effectively, in this case, no data is retrieved into the program's address space, but the program can still examine the status returned in the *vtqlca.vtqlcode* variable, which may

[1] Analogous to the embedded SQL WHENEVER statements.

indicate that the statement was successfully executed, that no qualifying tuples were found or that an error occurred.

The first argument of the `evtql_select` function is a structure describing the `SELECT` statement, together with the host language variables that are used in the statement (except the ones used in the `INTO` clause). Each field of the structure corresponds (roughly) to a clause of the `SELECT` statement. If a clause contains a comma-separated list of values (such as the select list or the `FROM` clause), an array is used to represent the clause within the structure, and each slot of the array is filled with the appropriate value. This arrangement eliminates the need for parsing during run time. For each expression in which host language variables may be used (i.e. the expressions in the `SELECT` list, the `WHERE` clause and the `HAVING` clause), a supplemental array is provided, containing variable descriptions and pointers indicating the offset, within the expression, where the host language variable must be inserted. This translation is demonstrated in Figure 2[2]:

```
/* EXEC VTQL SELECT EmpName, Salary, Period INTO :name, :salary, :period
FROM Employees E1 WHERE EmpId = :empid and BEGIN(VALID(E1)) = :targetDate; */

static evtql_select_struct vtql_sel_001 = {
{
   /* Number of items in select list plus a description for each item. Since no host
   variables are used here, the corresponding lists are empty. */
   {3, {{"EmpName", {0}}, {"Salary", {0}}, {"Period", {0}}}},
   /* The FROM clause */
   {1, {"Employee", "E1"}},
   /* The WHERE clause, including the description of the two host variables. Each description
   contains the offset within the string, where the variable value must be inserted. Notice
   the translation of VALID(E1) to E1.VT. */
   {"EmpId =  and BEGIN(E1.VT) = ",
        {2, {VTQL_CHAR, 5, NULL, 8}, {VTQL_DATE, 10, NULL, 29}}}
};

/* Structure describing the INTO clause. */
static evtql_varlist vtql_varlist_001 = {
3, {{VTQL_CHAR, 20, NULL}, {VTQL_INT, 4, NULL}, {VTQL_INTERVAL, 25, NULL}}
};

/* data structure initialisation */
vtql_sel_001.where.vars[0].ptr = empid;
vtql_sel_001.where.vars[2].ptr = targetdate; vtsl_varlist_001.vars[0].ptr = name;
vtql_varlist_001.vars[1].ptr = &salary; vtql_varlist_001.vars[2].ptr = period;

/* call to the library */
evtql_select(&vtql_sel_001, &vtql_varlist_001);
```

Figure 2 - Translation of a singleton SELECT statement

The first comment in Figure 2 shows the original EVTQL statement. The following two declarations are generated by the EVTQL preprocessor and are inserted at the beginning of the output file, whereas the executable statements that follow are inserted at the place where the embedded VTQL statement was located in the original file. The first group of statements

[2] A number of fields which are not used in the example is omitted (e.g. the field describing the `GROUP BY` clause) for brevity reasons, but they are actually emitted by the preprocessor. Additionally, arrays are initialised in separate declarations, but in this example their initialisation is merged with the declaration of the data structure which includes them.

fixes the pointers to the host variables, which are contained in the data structures. Data structure declaration cannot include pointer initialisation, because host variables may be either local or global, and data structures are declared at global level. Declaring the data structures at global level is required, since the declarations include initialisations, and many compilers reject such declarations at local (function) level. Additionally, using global declarations avoids unnecessary initialisation upon every entry to the function. Data structures are declared with *static* storage class, to avoid clashes with variables in other preprocessed files.

B. The INSERT, DELETE and UPDATE statements

Data insertion and the non-cursor forms of the `DELETE` and `UPDATE` statements are handled by calls to EVTQL library functions. Data deletion and update is performed by the `evtql_delete` and `evtql_update` functions, respectively, whereas for data insertion, one of the functions `evtql_insert_values` and `evtql_insert_query` is called, depending on whether the data to be inserted are specified via the `VALUES` clause or a query. In all cases, data structures formulation follows the rules described for the `SELECT` statement. Each function call is supplied with parameters that describe the requested operation.

C. The DECLARE CURSOR statement

The `DECLARE CURSOR` statement is not directly translated to a call to the EVTQL library. The preprocessor memorises the cursor name and the associated `SELECT` statement, and this data are used when the corresponding `OPEN` statement is processed. At this point, however, the `SELECT` statement is syntactically analysed, errors (if any) are reported and appropriate data structures are formulated in the preprocessor's memory.

To each cursor that is declared in a source file, the EVTQL preprocessor assigns a *DBMS statement name* and a *DBMS cursor name* (these are unique, system-generated identifiers). Additionally, for each declared cursor the preprocessor generates four functions, which are appended to the output file. The first function is named `evtql_open_CName` (*CName* is the name of the VTQL cursor) and contains the following embedded SQL statements:

```
EXEC SQL PREPARE dbms_statement FROM :host_string;
EXEC SQL DECLARE dbms_cursor CURSOR FOR dbms_statement;
EXEC SQL OPEN dbms_cursor;
```

(*dbms_cursor* and *dbms_statement* are the DBMS cursor name and statement name which are assigned to the cursor by the EVTQL preprocessor; *host_string* is a parameter passed to the function.) This function is called by the `evtql_open_cursor` EVTQL library function (presented later) to open the DBMS cursor, because embedded SQL packages do not allow cursor declarations via dynamic statements, or declarations in which the cursor name is stored in a host language variable.

The following two functions, namely `evtql_fetch_CName` and `evtql_close_CName`, are generated for analogous reasons. `evtql_fetch_CName` contains the embedded SQL statement

```
EXEC SQL FETCH dbms_cursor INTO :descriptor;
```

where *descriptor* is a parameter of type *DBMS_descriptor*, which is passed to the function. Function `evtql_fetch_CName` is called by the `evtql_fetch` and `evtql_fetch_descriptor` EVTQL library functions, when data fetching through the VTQL cursor is required. The `evtql_close_CName` function contains the embedded SQL statement

```
EXEC SQL CLOSE dbms_cursor;
```

and is called by the `evtql_close_cursor` EVTQL library function.

The fourth function generated for each declared cursor is named `evtql_exec_CName`, and contains the embedded SQL statement

```
EXEC SQL EXECUTE IMMEDIATE :cursor_statement;
```

This function is called whenever deletions or updates must be performed through the RDBMS cursor and is used because some DBMSs (e.g. Ingres [12]) require that all operations through a cursor must be performed from within the file that the cursor has been declared. *cursor_statement* is a dynamically formulated SQL statement (by some function in the EVTQL library), which is passed as a parameter to the `evtql_exec_CName` function.

In order to maintain the association between the EVTQL cursor name and the elements constructed by the EVTQL preprocessor (the DBMS statement name, the DBMS cursor name and the four functions) during run-time, the EVTQL preprocessor must be invoked in a special mode. When the EVTQL preprocessor operates in this mode, it processes all embedded VTQL source files compiles in a separate file a *table of cursors*, which is a host language array

of records, containing an entry for every declared cursor. Each entry contains the EVTQL cursor name, the DBMS cursor and statement names, plus pointers to the four preprocessor-generated functions. This file must be processed by the host language compiler to produce an object file, which should be linked into the temporal application executable file.

D. The OPEN statement

The `OPEN` statement prepares a cursor so that it can be used for data fetching. It is translated by the EVTQL preprocessor to a call to the `evtql_open_cursor` EVTQL library function. The function accepts two arguments, the first one being the name of the cursor, whereas the second argument is a structure describing the `SELECT` statement. If the `SELECT` statement includes references to host language variables, then appropriate assignment statements are inserted before the call to the EVTQL library function, so as to adjust the pointers to these variables.

E. The FETCH statement

The EVTQL `FETCH` statement is translated by the preprocessor to a call to the `evtql_fetch` EVTQL library function. The function is supplied with two parameters, the first one being the cursor name, whereas the second parameter is an array of host language variable descriptors. Each descriptor contains the type and the size of each variable, and a slot for the pointer to the actual memory location of the variable. This slot is filled by assignment statements which are inserted by the preprocessor, before the call to the `evtql_fetch` EVTQL library function.

F. The DELETE statement (cursor form)

The cursor form of the EVTQL `DELETE` statement is translated by the EVTQL preprocessor to a call to the `evtql_delete_cur` library function. The function is supplied with three parameters, the first two being the cursor name and the name of the target table, respectively. The third parameter describes the *VT_selection*, and its structure is identical to the respective parameter used in the non-cursor form of the `DELETE` statement. If the *VT_selection* syntactic construct is specified and the right hand side expression of the assignment contains references

to host language variables, then appropriate assignment statements are inserted before the call to the EVTQL library function, so as to adjust the pointers to these variables.

G. The UPDATE statement (cursor form)

The cursor form of the EVTQL `UPDATE` statement is translated by the EVTQL preprocessor to a call to the `evtql_update_cur` library function. Four parameters are passed to the library function, the first two being the name of the cursor, the name of the table to be updated, respectively. The third and fourth parameters describe the *VT_selection* syntactic construct and `SET` clause, respectively. If these clauses contain references to host variables, then appropriate assignment statements are emitted by the preprocessor, in order to adjust the pointers to the variables.

H. The CLOSE statement

The `CLOSE` statement is translated by the preprocessor to a call to the VTQL library function `evtql_close_cursor`, which is provided with a single parameter, designating the name of the cursor to be closed.

3.1.2 Implementing the WHENEVER statements

The `WHENEVER` statement may be used in an embedded VTQL program to simplify the database error handling process. The EVTQL preprocessor supports three flavours of the `WHENEVER` statement, which may be used to specify the action to be taken in the event of an error (`VTQLERROR`), a warning (`VTQLWARNING`) or when a select statement could not return any data (`NOT FOUND`). If a source file contains `WHENEVER` statements, then the VTQL preprocessor automatically inserts after the code implementing each translated statement appropriate host language statements which test the value of the *vtqlca.vtqlcode* variable and perform the designated action as needed. This procedure is illustrated in Figure 3.

```
/* EXEC VTQL WHENEVER VTSQLERROR GOTO error_handler; */
/* EXEC VTQL WHENEVER VTSQLWARNING CONTINUE; */
/* EXEC VTQL WHENEVER NOT FOUND PERFORM no_more_data_dialog; */

/* EXEC VTQL SELECT Name, Salary INTO :name, :salary FROM employee; */
evtql_select(...);
if (vtqlca.vtqlcode < 0) /* VTQLERROR */
       goto error_handler;
/* Since the "continue" action is specified for the VTQLWARNING
event, no tests are performed for this event class */
if (vtqlca.vtqlcode == VTQL_NO_MORE_DATA) /* NOT FOUND */
       no_more_data_dialog();
```

Figure 3 - Implementation of the WHENEVER statements

3.2 The EVTQL Library

The EVTQL library is a collection of procedures which implement the embedded VTQL statements. In the following paragraphs, the algorithms employed by these procedures to perform the requested operations, are described. For brevity reasons, details of reporting the status of the operations' execution to the application are not discussed; this is always done by means of the *vtqlca.vtqlcode* status variable, which the procedures set to 0, for successful completion, or to a non-zero value, to indicate an error.

3.2.1 The SELECT statement ("singleton" form)

The singleton selection operation is performed via the `evtql_select` library procedure, which accepts two arguments, the first one describing the `SELECT` statement, whereas the second argument describes the variable list into which the values will be fetched. The `evtql_select` library procedure processes the request using the following algorithm:

1. If the `SELECT` statement is *snapshot-transformable* (see page 3), the `evtql_select` library procedure reconstructs the original select statement and prepares a *descriptor* (a DBMS-specific data structure) which is actually a list of variable descriptions (type, size and memory address). The prepared descriptor contains one entry for each variable that appears in the original `INTO` clause. The data pointer of each entry is set to the memory location of the host language variable, so that the data will be directly placed into the desired memory area, with no additional data copying. The reconstructed `SELECT` statement is executed and the variables are fetched into the host language variables using the following embedded SQL statement:

```
EXEC SQL PREPARE dbms_stmt FROM :stmt;
EXEC SQL DECLARE tmp_cursor CURSOR FOR dbms_stmt;
EXEC SQL OPEN tmp_cursor;
EXEC SQL FETCH tmp_cursor USING DESCRIPTOR :d1;
EXEC SQL EXECUTE IMMEDIATE :stmt USING DESCRIPTOR :d1;
```

   where the host variable *stmt* contains the reconstructed statement and *d1* is the prepared descriptor.

2. If the query is not snapshot-transformable, the EVTQL library uses the algorithms employed by the terminal monitor application to evaluate the query. Results, however, are not output to the screen, but stored in a temporary table *tmp_table*. Finally, the batch of

embedded SQL statements described in case (1) is executed to fetch the data into a suitably prepared descriptor. In this case the host variable *stmt* contains the SQL statement

`SELECT * FROM tmp_table;`

Finally, the intermediate table is dropped.

3.2.2 The INSERT statement and the non-cursor versions of the DELETE and UPDATE statements

Data insertion is implemented through two functions in the EVTQL library, namely `evtql_insert_values` and `evtql_insert_query`, which are invoked when the data to be inserted are specified by means of the `VALUES` clause or an select subquery, respectively. Data deletion and update are handled by the `evtql_delete` and `evtql_update` EVTQL library functions. In all cases, the EVTQL library functions employ the algorithms used by the terminal monitor application to handle the user request.

3.2.3 The OPEN statement

When a cursor is *opened* using the `EXEC VTQL OPEN` statement, the execution of the VTQL `SELECT` query which is associated with the statement is actually triggered. The `evtql_open_cursor` EVTQL library procedure distinguishes three cases for the processing of the request:

1. *the `SELECT` statement is snapshot-transformable and results either to a non-updatable tuple set, or to an updatable tuple set, which is not derived from a valid time table.* In this case, the original statement is reconstructed into a host language string variable, and the preprocessor-generated `evtql_open_CName` function is called. The reconstructed statement is used as an argument to this call. This invocation triggers the preparation of a statement and the opening of a cursor at the DBMS level.

   The procedure completes by modifying the relevant entry in the *table of cursors*. Flags are set to designate that the cursor is currently open, whether it points to an updatable tuple set or not and the current status of the cursor, which is initially set to *`NO_CURRENT`* (no tuple is current for the cursor). If the tuple set is updatable, an additional flag is set within the entry, indicating that the tuple set is not derived from a table with valid time semantics.

2. *the* `SELECT` *statement is snapshot-transformable and results to an updatable tuple set, which is derived from a table with valid time semantics.* In this case, the EVTQL library constructs into a host language string variable an SQL query, which is identical to the original one but its `SELECT`-list is extended to include all columns which are part of the table's schema, but are not listed in the original select list. These extraneous columns are necessary for the implementation of the cursor forms of the `DELETE` and `UPDATE` statements, which are described later. The steps of opening the RDBMS cursor and registering the entry in the table of open cursors are identical to the ones described for case (1), except that the "result set updatability" and "valid time semantics" flags are set to `TRUE`. Additionally, memory for fetching the extra columns is allocated in the EVTQL library's private memory pool.
3. *the* `SELECT` *statement is not snapshot-transformable.* In this case, the `evtql_open_cursor` procedure applies the algorithms used in the terminal monitor application to evaluate the query, but stores the results in a temporary table *tmp_table*, rather than displaying them on the screen. The query

   `SELECT * FROM tmp_table`

   query is stored into a host language string, and passed to the preprocessor-generated function `evtql_open_CName`, so as to open the DBMS-level cursor. Finally, the appropriate flags are set in the EVTQL library's table of cursors, as in case (1). However, in this case the "result set updatability" flag is set to `FALSE` (since the operations query involves at least one set operation that renders the query result not updatable), and the name of the temporary table is stored in the cursor's entry.

3.2.4 The FETCH statement

When a `FETCH` command is processed, the `evtql_fetch` function locates into the table of cursors the entry for the cursor through which data fetching is requested and verifies that it is open. Afterwards, data are fetched through the DBMS-level cursor into the EVTQL library's memory calling the preprocessor-generated `evtql_fetch_CName` function. An appropriately prepared descriptor is passed as an argument to this function. The fetched data are subsequently copied into the variable list designated by the second parameter of the

`evtql_fetch` function (note that extraneous columns that are fetched for updatable tuple sets that are derived from tables with valid time semantics are not copied). If no more data can be fetched (as indicated by the *sqlca.sqlcode* status variable), the status field of the cursor entry is set to *`DATA_EXHAUSTED`*, and the error is reported to the application through the *vtqlca.vtqlcode* status variable. If, however, data fetching is successful, the cursor status is set to *`HAS_CURRENT`* and control is returned to the application.

3.2.5 The DELETE statement (cursor form)

The `evtql_delete_cur` EVTQL library function initially examines if the cursor points to an updatable tuple set. If the result set is not updatable or the cursor status is found to be different than *`HAS_CURRENT`*, control is immediately returned to the application. If both tests succeed, then the second parameter is examined to determine if the *VT_selection* syntactic construct was used in the original statement, designating that only *part of the tuple* is to be removed. If this syntactic construct was not specified, the whole tuple is deleted by formulating into a host language string variable the embedded SQL statement

```
DELETE FROM table_name WHERE CURRENT OF dbms_cursor_name;
```

and passing it to the preprocessor-generated `evtql_execute_CName` function. If, however, the *VT_selection* syntactic construct was specified, it is verified that table *table_name* actually has valid time semantics (an error is raised if this is not the case) and one of the following actions is taken, depending on the relative position of the tuple's valid time (denoted as T_{vt}) and the value of the expression used in *VT_selection* (denoted as T_{del}):

1. if T_{del} *contains* T_{vt}, then the whole tuple is deleted from the table.
2. if the time points included in T_{vt} but not in T_{del} are consecutive, then the tuple's valid time is set to the difference T_{vt} - T_{del}, by building the statement
   `UPDATE TableName SET valid_column = T new WHERE CURRENT OF dbms_cursor_name`
   into a host language string variable and calling the function `evtql_execute_CName` (T_{new} is set to T_{vt} - T_{del}).
3. if the time points included in T_{vt} but not in T_{del} are not consecutive, requiring thus two periods for their representation (denoted as p_1 and p_2), then the tuple's valid time is set to p_1, as described in case (2) and a new tuple is inserted in the target table. All columns of the

new tuple are equal to the modified tuple's corresponding columns, except for the valid time, which is set to p_2.

4. if T_{vt} and T_{del} do not overlap, no action is taken.

In all cases, the status field of the cursor entry is set to `NO_CURRENT`, disallowing further deletions and/or updates, until the next tuple of the result set is fetched.

3.2.6 The UPDATE statement (cursor form)

Analogously to the `evtql_delete_cur` function, the `evtql_update_cur` EVTQL library function initially verifies that the cursor actually points to an updatable tuple set and that a current tuple exists, terminating prematurely if any of these conditions is not met. If both checks succeed, one of the following actions is taken, depending on whether the *VT_selection* syntactic construct was used in the original statement:

1. *VT_selection was not used.* In this case, the EVTQL library examines if the target table has valid time semantics or not, and takes one of the following actions:
   A. *the table does not have valid time semantics*. In this case, the current tuple is updated, as specified in the `SET` clause, by using the cursor form of the embedded SQL `UPDATE` statement (an appropriate `UPDATE` statement is formulated in a host language string and passed to the `evtql_execute_CName` function). Errors reported by the DBMS are trapped by the EVTQL library and reported back to the application.
   B. *the table has valid time semantics.* In this case, the current tuple is removed from the table, the new values of the columns designated in the `SET` clause are computed, and the updated version of the current tuple is constructed. If *value-equivalent tuples* ([13]) with overlapping valid timestamps are not allowed for the updated table, the table is scanned for tuples whose data columns are identical to the updated tuple's corresponding columns and whose valid time overlaps with the updated tuple's valid time. If such a tuple exists, the update violates this constraint, thus changes to the database state made by this statement are undone and control is returned to the application.

If no such tuple is found, the update procedure continues normally. In the case that valid time tables are not stored in a coalesced form, the tuple may be flushed to the database immediately. If, however, valid time tables are not stored in a coalesced format, the updated tuple *can not* be inserted into the table at this point, since this insertion might have the effect of altering the values of a tuple in the result set which has not been fetched through the cursor yet. In other words, there may exist a tuple *res_tuple* in the result set that has not been yet processed and can be coalesced with the updated tuple (*upd_tuple)*. Thus, inserting the tuple at this point will lead to replacing *res_tuple* with the result of *COALESCE*(*res_tuple*, *upd_part*), which is undesirable. Instead of inserting the updated tuple in the table at this point, it is stored in a *pending tuple list*, which is associated with the cursor. These tuples are inserted to the database (and possibly coalesced with other tuples) when the cursor is closed. However, the list is scanned when the violation of primary key uniqueness is checked, to ensure that an update does not clash with previous modification operations.

In both cases, the update operation completes by setting the status flag of the cursor to `NO_CURRENT`, disallowing any further updates through this cursor until the next tuple is fetched.

2. *VT_selection was used.* In this case, it is verified that the updated table is a valid time table, and if it is not, control is immediately returned to the application with an appropriate error indication. Subsequently, the part of the tuple which is subject to update is removed from the database, following the procedure described for the cursor version of the `DELETE` statement (paragraph 3.2.5). The removed part of the tuple is modified as specified in the `SET` clause and, primary key uniqueness checks are conducted as needed (see case (B) above) and the updated tuple is either flushed to the database or inserted in the pending tuple list of the cursor, depending on whether a coalesced storage scheme is used for valid time tables. Finally, the status flag of the cursor is set to `NO_CURRENT`, disallowing further updates through this cursor until the next tuple is fetched.

3.2.7 The CLOSE statement

Cursor closing is handled by the `evtql_close_cursor` EVTQL library function, which closes the DBMS-level cursor (by calling the `evtql_close_CName` function, which has been created by the preprocessor), destroys the temporary table created during the `SELECT` statement's evaluation (if any), frees memory that was used for storage of extraneous columns (for updatable results) and removes the corresponding entry from the EVTQL library's table of open cursors. If the list of pending tuples associated with the cursor is not empty, then the tuples in the pending list are inserted in the table, performing any necessary coalescing. Constraint violation checks (for value equivalent tuples with overlapping valid timestamps) need not be conducted at this point, since all tuples in the pending list have been checked when the corresponding `UPDATE` statement was processed.

4. Conclusions-Future Work

In this paper, we presented an embedded temporal language, and proposed an architecture for its implementation, on top of a relational DBMS. The software modules comprising the development environment were presented (the *preprocessor* and the *EVTQL library*), and the algorithms employed by the software modules were discussed. Future work will focus on the development of the EVTQL preprocessor and the EVTQL library for the TSQL2 language. Optimisation issues will be also studied, along with techniques for enhancing concurrency control within multi-user environments.

5. References

[1] A. V. Aho, R. Sethi and J. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1985.

[2] G. Ariav, "Information Systems for Managerial Planning and Control: A Conceptual Examination of their Temporal Structure", Journal of MIS, vol. 8, 1992.

[3] M. Böhlen, "Temporal Database System Implementations", SIGMOD RECORD Vol. 24, No. 4, December 1995.

[4] M. Böhlen, "TimeDB Software", Department of Mathematics and Computer Science, Aalborg University, 1995.

[5] R. Chandra and A. Segev, “Using Next Generation Databases to Develop Financial Applications”, Proceedings of the First International Conference on Application of Databases (ADB), Vadstena, Sweden, June 1994.

[6] J. Clifford, C. Jensen, R. Snodgrass, M. Böhlen, H. Dewand and D. Schmidt, “Panel: The State-of-the-Art in Temporal Database Management: Perspectives from the Research and Financial Application Communities”, Proceedings of the VLDB International Workshop on Temporal Databases, Zurich 1995.

[7] C. Combi, F. Pinciroli, G. Musazzi and C. Ponti, “Managing and Displaying Different Time Granularities of Clinical Information”, Eighteenth Annual Symposium on Computer Applications in Medical Care, Washington DC, U.S.A. 1994.

[8] C. Combi, F. Pinciroli, M. Cavallaro and G. Cucchi, “Querying Temporal Clinical Databases with Different Time Granularities: The GCH-OSQL Language”, Nineteenth Annual Symposium on Computer Applications in Medical Care, New Orleans, U.S.A., 1995.

[9] C. J. Date and H. Darwen, “A Guide to the SQL Standard” (third edition), Reading, Massachusetts, Addison-Wesley, 1993.

[10] C. J. Date, “An Introduction to Database Systems” (sixth edition), Addison-Wesley Publishing Company Inc., 1995.

[11] R. Hunter, “Compilers: Their Design and Construction Using Pascal”, Wiley and Sons, 1985.

[12] Ingres Corporation, “Ingres SQL and ESQL Reference Manual” (for release 6.4), 1991.

[13] C. S. Jensen, J.Clifford, R. Elmarsi et al. "A Consensus Glossary of Temporal Database Concepts", SIGMOD Record, 23(1), pp. 52-64, March 1994.

[14] Oracle Corporation., “SQL Language Reference Manual” (for release 7), 1993.

[15] Oracle Corporation, “ORACLE8 Time Series Cartridge User’s Guide”, November 1997.

[16] ORES Project (ESPRIT III P7224) Deliverable D4.2: “Application Manual”, edited by 01 Pliroforiki, University of Athens, Agricultural University of Athens and Information Dynamics, 1993. Available at ftp://ftp.di.uoa.gr/pub/ores/rep/d4_2.ps.gz.

[17] D. Schmidt and R. Mari, “Time Series, a Neglected Issue in Temporal Database Research?”, Proceedings of the VLDB International Workshop on Temporal Databases, Zurich 1995.

[18] Sybase Inc., "Transact SQL User's Guide" (for release 10), 1994.

[19] B. Theodoulidis, A. Ait-Braham and G. Karvelis, "The ORES Temporal DBMS and the ERT-SQL Query Language", Proceedings of the 5th International Conference on Database and Expert System Applications, Athens 1994.

[20] K. Torp, C. Jensen and M. Böhlen, "Layered Implementation of Temporal DBMSs-Concepts and Techniques", TimeCenter Technical Report TR-2, 1997 available from http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/publications.html

[21] K. Torp, C. Jensen and R. T. Snodgrass, "Stratum Approaches to Temporal DBMS Implementations", TimeCenter Technical Report TR-5, 1997, available from http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/publications.html

[22] The TSQL2 Language Design Committee, "The TSQL2 Temporal Query Language", Edited by R.T. Snodgrass, Kluwer Academic Publishers, 1995.

[23] C. Vassilakis, P. Georgiadis and A. Sotiropoulou, "A Comparative Study of Temporal DBMS Architectures", Proceedings of the 7th International DEXA Workshop, pp. 153-164, Zurich, 1996.