An SQL Interface for Common Lisp[1]

Alan S. Gunderson

Advanced Technology Research and Development Group

Digital Equipment Corporation
290 Donald Lynch Blvd
Marlborough MA 01752

January 1991

Abstract

Access to persistent database storage from Common Lisp applications is an increasingly frequent requirement. This paper discusses a software module that allows a Common Lisp program to access a relational database management system. Database queries are expressed as SQL statements. In Common Lisp, the SQL query is represented as a string, which allows sequence functions to be used to construct the query. Through a foreign function interface, the SQL query is passed from Lisp to a C component. The C component contains embedded database statements to utilize the dynamic SQL facility provided by the database management system. The database result table is returned to Lisp. Functions are provided to convert the returned data into a list of structures. Each structure corresponds to one row in the database result table. Structure accessor functions allow each individual attribute (field) in the structure representation of a row in the result table to be obtained.

[1] This paper describes one phase of research being carried out in the Digital Equipment Corporation Advanced Technology Research and Development Group.

© 1991 Alan S. Gunderson, Digital Equipment Corporation

1 Introduction

The SQLCL module allows a Common Lisp [Steele 1990] program to access a relational database management system (DBMS). Database queries are expressed as SQL statements. In Common Lisp, the SQL query is represented as a string, which allows the sequence functions to be used to construct the query. Through a foreign function interface, the SQL query is passed from Lisp to a C component. The C component contains embedded database statements to utilize the dynamic SQL facility provided by the DBMS. The database result table is returned to Lisp. Functions are provided to convert the returned data into a list of structures. Each structure corresponds to one row in the database result table. Structure accessor functions allow each individual attribute (field) in the structure representation of a row in the result table to be obtained. SQLCL is implemented in Lucid Common Lisp[1] and C and uses the ULTRIX/SQL[2] DBMS on a Digital DECstation 3100 RISC workstation running the ULTRIX Version 4.0 operating system.

2 Related Work

The CLING (Common Lisp/INGRES Interface) from UC Berkeley [Sedayao and Irwin 1987] provided a QUEL[3] [Stonebraker et al., 1976] interface for Common Lisp. With CLING, access to the database occurred through calls to the INGRES object library. In contrast, SQLCL uses the dynamic SQL facility found in ULTRIX/SQL and other relational DBMSs. This should increase the longevity of the SQLCL approach, as the implementation is not dependent upon functions in the DBMS object library.[4] Direct calls to the object library are probably faster than the use of dynamic SQL. For our applications, database access speed was not a concern and we never found the performance of our dynamic SQL approach to be at all bothersome. For an application concerned with large numbers of transactions per second, the same may not be true.

Ideas for the use of Dynamic SQL were obtained from the example program called "The ULTRIX/SQL Terminal Monitor Application" that is described in Appendix B of the ULTRIX/SQL Companion Guide For C manual. It would be fair to characterize the C component of SQLCL as a significant enhancement and evolution of the ULTRIX/SQL Terminal Monitor Application, including a port of the program to the more stringent requirements for C programming on a RISC processor, e.g., more attention must be paid to correct word alignment

[1] Lucid and Lucid Common Lisp are trademarks of Lucid, Inc..
ULTRIX and DECstation are trademarks of Digital Equipment Corporation.
INGRES is a trademark of INGRES Corporation.

[2] The relational DBMS is ULTRIX/SQL. ULTRIX/SQL is the designation used by Digital for the INGRES DBMS. ULTRIX/SQL is a software layered product for Version 4.0 of the ULTRIX operating system. Version 1.0 of ULTRIX/SQL is based on INGRES Release 6.2.

[3] QUEL is an older INGRES query language based upon the tuple-oriented relational calculus.

[4] Initially, we had hoped to use CLING. Unfortunately, the INGRES object library had undergone considerable change since the CLING system was written and CLING no longer functioned. Also, we wanted to use SQL rather then QUEL.

when casting data among data types and to their underlying architectural representations.[1] Section 3.3 discusses the word alignment problem in more detail.

The PCLOS system [Paepcke 1989] is a detailed approach to object persistence for an object-oriented language. Several dimensions of the object persistence problem are analyzed. One dimension is the object mode - the way for an application program to control the location and sharability of objects at run-time. One such object mode is called "uncached operation" and is described in the paper as follows:

> "All accesses to the persistent object will by default involve database operations. A read will fetch a value by submitting a query which may be precompiled for improved efficiency. A write will cause a database update operation."

A system such as SQLCL could be used for the queries and update operations in PCLOS.

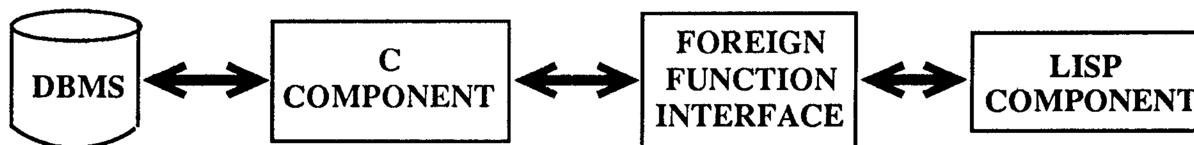
Work at GTE Laboratories on the Distributed Object Manager (DOM) [Manola 1990] as a building block for a knowledge-based integrated information system is another comprehensive approach that is being pursued. Applications can interact with other application through DOMs. A local-application interface (LAI) provides a mapping between an application and a DOM. An LAI can accept messages, such as a data request, and translate them into local request to an application for processing. If the application is a DBMS, the LAI has several similarities to the SQLCL system.

3 SQLCL Design

The design of SQLCL is comprised of three components:

- The Lisp component. This includes the SQLCL interface functions for the Lisp programmer and a data transformation utility for converting data collected from the database into a form more useful for the Lisp programmer to utilize in his application.
- The C component. A facility of the DBMS called dynamic SQL is accessed via a C program. Dynamic SQL is like the Lisp **eval** function, except SQL statements are evaluated.
- The foreign function interface. This allows function calling between Lisp and C components.

The following figure illustrates the interaction among the components.



Ensuing subsections will discuss the SQLCL components in more detail.

[1] Oh, the joys of C programming after years in the comfort of Lisp.

3.1 The Lisp Component

Six Lisp function comprise the interface that the Lisp programmer utilizes to access the DBMS and to transform the information retrieved from the DBMS. These functions in the Lisp component call functions in the C component through the foreign function interface. The C component accesses the DBMS and may generate output to the user or return information back to the Lisp component, again through the foreign function interface. The functions in the Lisp component are:

- open-db
- close-db
- interactive-db
- sql-query-print-db
- sql-query-db
- structify-tuple-data

A format similar to that used in CLtL [Steele 1990] will be used to describe the interface functions.

open-db *dbname &key (interactive-dbname-query-flag 0)* [Function]

A DBMS can support numerous databases, where a database is a collection of relational tables that may be semantically related through common key attributes among the tables. One of the databases managed by the DBMS must be opened before SQL queries can be posed to the DBMS to manipulate or retrieve information in the relational tables. The **open-db** function allows a database to be opened by the Lisp programmer.

The **open-db** function returns T if a connection to the database specified in the *dbname* argument can be established. The *dbname* argument should be a string. The special variable **current-db-name** is also set to *dbname* to record the current database name and to indicate that a database is open. If a database is already opened or an error occurs during the process of opening the database, an error is signaled.

When the *interactive-dbname-query-flag* keyword argument is non-zero, the *dbname* argument is ignored. Instead, the user is prompted for the name of a database for which a connection is desired.

The **open-db** function also supports a "lazy" loading capability for both the object code of the C component of SQLCL and the ULTRIX/SQL and C object libraries.[1] An "eager" loading of the object code could occur when an application using SQLCL is loaded into the Lisp environment. Unfortunately, the loading of the object code takes almost 2 minutes and is wasted time if

[1] The C component of SQLCL is compiled into an object file. As the C code needs to work with the DBMS, a language called "embedded SQL" is used in the C program. A preprocessor is used on the C program containing the 'embedded SQL" statements to convert them into standard C code and function calls to the ULTRIX/SQL and C libraries. The C object code and libraries are then loaded into Lisp and dynamic linking occurs between the function calls in the C component and the function object code in the libraries.

SQLCL is not used.[1] It is possible to force an "eager" load, e.g., when building a production version of a Lisp application.

close-db [Function]

When a Lisp programmer is done utilizing a particular database managed by the DBMS, the database should be closed with the **close-db** function. The **close-db** function returns T if the connection to an open database is successfully closed. The special variable ****current-db-name**** is reset to NIL to record the fact the that no database is open. If there is no database to close (one was never opened), the function returns NIL.

interactive-db [Function]

Most DBMSs provide an interactive utility where the user can enter query statements and have the results of the query displayed back to the user.[2] Such utilities provide a nice environment for formulating and debugging complex query statements prior to coding them in an application program. The **interactive-db** function provides such a utility to the Lisp programmer.

When the **interactive-db** function is called, an interactive SQL terminal monitor is invoked from Lisp. The user is prompted for the database to open. A loop is then entered where SQL queries can be typed and query results are printed.

sql-query-print-db *sql-query-str &key (dquote-strings-flag nil)* [Function]

This function is an intermediate step between accessing a database interactively with the **interactive-db** function and the **sql-query-db** function described next. The **sql-query-print-db function** is passed an SQL query. The results of the query are displayed to the user in the same format as that used in the **interactive-db** function. The database result information is pretty printed for the user rather than being returned to Lisp (the **sql-query-db** function described next *does* return the information to Lisp).

With **sql-query-print-db**, an SQL query, passed as a string in *sql-query-str*, is sent to the database for dynamic SQL processing. The results of the query are printed to standard-output and T is returned when the SQL statement is successfully processed. The output is generated by functions in the C component. If the *dquote-strings-flag* keyword argument is passed a non-NIL value, string attributes in the database are printed with surrounding double quotes. Any errors that occur in the dynamic SQL processing of the SQL statement by the database will be printed to standard-output. The function returns NIL if the SQL statement can not be run. An error is signaled if no database is open.

Example:

```
Lisp> (sql-query-print-db "select o_id, o_name
                           from omap_node
                           where o_id < 10;"
                          :dquote-strings-flag t)
```

[1] During our development work, we found that we were infrequently using the DBMS via SQLCL relative to loading the system for debugging of other parts of our application software that did not require access to the database.

[2] A read-eval-print loop for users of the DBMS.

```
[1] o_id [2] o_name

[1] 1 [2] \"VAX-9000-RAINYA\"
[1] 2 [2] \"H4000-7\"
[1] 8 [2] \"THICKWIRE-COMPUTER_RM_SEGMENT\"
[1] 3 [2] \"XCVR-CABLE-7\"
[4 row(s)]
T
```

sql-query-db *sql-query-str &key (dquote-strings-flag nil)* [Function]

This function will be the "workhorse" in a Lisp application program that needs to access a database. A query is passed to **sql-query-db**, the database processes the query (Lisp --> Foreign Function Interface --> C Component --> DBMS) and the results of the query (called a database result table) are returned to Lisp (DBMS --> C Component --> Foreign Function Interface --> Lisp). Then, the data transformation utility in the Lisp component can be used to convert the returned database result table into a form more useful for the Lisp programmer.

With **sql-query-db**, an SQL query, passed as a string in *sql-query-str*, is sent to the database for dynamic SQL processing. The results are returned as a character string suitable for processing by the **read-from-string** function. Reading the returned character string will yield a list of lists. Each sublist is a row in the relational table that is the result of the query.

If the *dquote-strings-flag* keyword argument is passed a non-NIL value, string attributes in the database are returned with surrounding double quotes. The returned value is thus a character string that can contain embedded character strings. If character string attributes in the database are atomic (no spaces exist in the string), the default value NIL for *dquote-strings-flag* can be used to return non-quoted character strings from the database. The non-quoted character strings will become Lisp symbols when the returned string is processed by the **read-from-string** function.

Any errors that occur in the dynamic SQL processing of the SQL statement by the database will be printed to standard-output. The function returns the NULL character string (in C parlance) if the SQL statement can not be run. An error is signaled if no database is open.

Examples:[1]

```
Lisp> (sql-query-db "select o_id, o_name
            from omap_node
                  where o_id < 10;")
"(([1] 1 [2] VAX-9000-RAINYA )
  ([1] 2 [2] H4000-7 )
  ([1] 8 [2] THICKWIRE-COMPUTER_RM_SEGMENT )
  ([1] 3 [2] XCVR-CABLE-7 ))"

Lisp> (sql-query-db "create table employee(
                  emp_id i4 with null,
                  emp_name varchar(50) with null);")
""
```

[1] The output strings are pretty-printed in list notation for readability.

```
Lisp> (sql-query-db "insert into employee(emp_id, emp_name)
                     values (157, 'Fred Smith');")
""

Lisp> (sql-query-db "insert into employee(emp_id, emp_name)
                     values (199, 'Jane Jones');")
""

Lisp> (sql-query-db "select * from employee;" :dquote-strings-flag t)
"(([1] 157 [2] \"Fred Smith\" )
 ([1] 199 [2] \"Jane Jones\" ))"
```

structify-tuple-data *data-string struct-name slot-name-list* [Function]

The database result table returned to Lisp as a string can be converted into a list of DEFSTRUCT structures. This transformation yields a database result table in a format that is very convenient and familiar to a Common Lisp programmer. Each structure in the list corresponds to a row in the database result table. The columns in a row can be retrieved using the structure slot accessor functions. The columns (also called attributes or fields) in a relational table are referred to by the attribute names in SQL queries, e.g., EMPLOYEE.EMP_ID. A structure accessor function, e.g., EMPLOYEE-ID, provides a similar interface to the result table attributes.

With the **structify-tuple-data** function, the *data-string* argument is passed a string returned from ULTRIX/SQL to Lisp via use of the **sql-query-db** function. The *struct-name* argument is the name designator for the structure and *slot-name-list* is a list of the structure slot names. A one-to-one mapping between the columns in the database result table and the slots in the structure should be made. The slot names do not have to be the same as the attribute names in the database but the number of items in the *slot-name-list* should correspond to the number of attributes (columns) in the database result table.

Examples:

```
Lisp> (setq tmp1 (sql-query-db "select * from employee;" :dquote-strings-flag t))
"(([1] 157 [2] \"Fred Smith\" )([1] 199 [2] \"Jane Jones\" ))"

Lisp> (setq tmp2 (structify-tuple-data tmp1 'employee '(id name-str)))
(#S(EMPLOYEE ID 157 NAME-STR "Fred Smith")
 #S(EMPLOYEE ID 199 NAME-STR "Jane Jones"))

Lisp> (employee-name-str (car tmp2 ))
"Fred Smith"

Lisp> (employee-id (car tmp2))
157
```

3.2 C Component for Dynamic SQL

This section will describe the C component of SQLCL. The use of dynamic SQL will be discussed in detail via examination of code fragments. In the C code, statements will appear preceded by "EXEC SQL". These are the "embedded SQL" statements that were briefly discussed

in a footnote in Section 3.1. Prior to compilation with the C compiler, a preprocessor, called **esqlc**, is used to convert the embedded "EXEC SQL" statements into standard C data structures and function calls to the ULTRIX/SQL and C object libraries.

For discussion purposes, assume the **sql-query-db** function discussed in Section 3.1 has been called in the Lisp component. The *sql-query-str* argument is passed an SQL statement as a string. Through the foreign function interface (see Section 3.4), the C function in the C component with the following proforma is called:

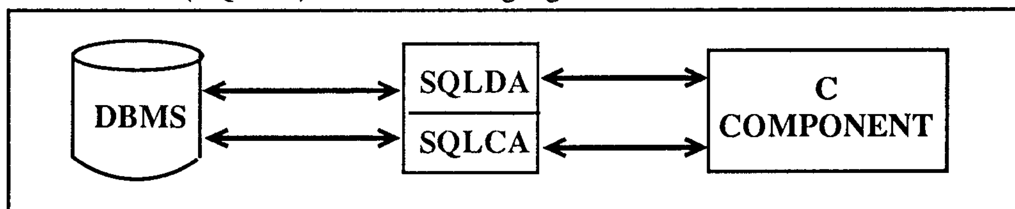
```
char * sqlcl_out_run_query(input_query, dquote_strings_flag)
   char *input_query;
   int dquote_strings_flag;
```

The *input_query* argument of the C function **sqlcl_out_run_query** is a pointer to the *sql-query-str* argument in the **sql-query-db** Lisp component function.

The function processing will now be described. The query is first run through a simple conversion routine and is then placed in the SQL statement input buffer data structure (*stmt_buf*). This C data structure is made visible to other "embedded SQL" statements in the C program via the declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
          char stmt_buf[STMT_MAX + 1]; /* SQL statement input buffer */
EXEC SQL END DECLARE SECTION;
```

The SQL statement in *stmt_buf* is now prepared[1] and described[2] into the SQL Descriptor Area (SQLDA). The SQLDA is a C structure used to communication type and size information about an SQL statement, between the C program and the DBMS. There is also another shared structure for information communication between the C program and the DBMS called the SQL Communication Area (SQLCA). The following figure illustrates this.



The code for the prepare and describe process is as follows:

```
/* Prepare and describe the statement. If we cannot fully
  describe the statement (our SQLDA is too small) then
  allocate a new one and redescribe the statement. */
EXEC SQL PREPARE stmt FROM :stmt_buf;
EXEC SQL DESCRIBE stmt INTO :sqlda;
if (sqlda->sqld > sqlda->sqln) {
   Init_Sqlda(sqlda->sqld);
   EXEC SQL DESCRIBE stmt INTO :sqlda; }
```

[1] The **prepare** statement encodes the dynamically constructed statement string for later execution.

[2] The **describe** statement retrieves type information about a prepared dynamic SQL statement.

Now that the SQL statement has been prepared and described, the sqld field in the SQLDA can be examined to determine the type of SQL statement. The type can be either an SQL selection statement,.e.g.,

```
select emp_id, emp_name from employee;
```

or a non-select SQL statement, e.g.,

```
create table employee( emp_id i4 with null,
                       emp_name varchar(50) with null);
```

With the statement type determination, appropriate routines can be invoked to process the SQL statement. This is show in the code below.

```
/* if 'sqld' = 0 then this is not a SELECT */
if (sqlda->sqld == 0) {
   EXEC SQL EXECUTE stmt;
   rows = sqlca.sqlerrd[2];
   total_output[0] = '\0'; }
else /* SELECT */
    rows = sqlcl_Execute_Select(dquote_strings_flag);
```

As shown in the last line in the previous code fragment, when the SQL statement is a select statement, the C function with the following proforma is called.

```
int sqlcl_Execute_Select(dquote_strings_flag)
   int dquote_strings_flag;
```

Several variables are first declared in this **sqlcl_Execute_Select** function

```
int rows;   /* Counter for rows fetched */

/* Ptr to total_output where retrieval results are collected */
char *tot_output_ptr;
```

Now, the **Print_Header** function is called.

```
/* Determine the result column names, allocate the result variables, and set up the types. */
Print_Header(0);  /* arg of 0 to only run the function for side effects */
```

The **Print_Header** function allocates a buffer for retrieving the data from the DBMS. The result buffer is a single character buffer whose size is determined by adding up the result column sizes for each column (attribute) of a row in the database result table. All the different database types are collapsed into integer (long), float (double precision), and character types.

Now the dynamic SQL statement is processed by the DBMS with the following statement. If an error occurs when running the SQL statement, a jump to Close_Csr occurs. The Close_Csr code recovers from the error and handles error reporting.

```
EXEC SQL WHENEVER SQLERROR GOTO Close_Csr;
```

After a dynamic SQL statement is processed, the dynamic cursor is opened.

```
/* Open the dynamic cursor */
EXEC SQL OPEN csr;
```

A dynamic cursor can be thought of as a row marker that can be moved forward through each row in a database result table. A cursor basically allows the full range of C and embedded SQL statements to utilize the data from the database a row at a time, i.e., the programmer has full control of the manipulation of the retrieved data.

Using the dynamic cursor, the query result data is now placed into the *total_output* character buffer as a list of lists. For example, the query

```
select emp_id, emp_name from employee;
```

yields the result table

emp_id	**emp_name**
157	Fred Smith
199	Jane Jones

which would be placed in the character buffer as:

```
"(([1] 157 [2] \"Fred Smith\" )([1] 199 [2] \"Jane Jones\" ))"
```

The code fragment below illustrates this process. Notice that the function **sqlcl_Print_Row** is called to produce the row data.

```
/* Fetch and gather the results for each row into total_output buffer */
tot_output_ptr = total_output;
rows = 0;
/* Add a left parenthesis  to indicate start of data, which will
  be a list of lists of data */
*tot_output_ptr++ = '\(';
 while (sqlca.sqlcode == 0)
 {
    EXEC SQL FETCH csr USING DESCRIPTOR :sqlda;
    if (sqlca.sqlcode == 0) {
      rows++;
     /* Add a ( to indicate start of row */
            *tot_output_ptr++ = '\(';
      sqlcl_Print_Row(&tot_output_ptr,
                  dquote_strings_flag);
      /* Add a ) to indicate end of row */
            *tot_output_ptr++ = '\)';
    }
 } /* While there are more rows */

/* Add a ) to indicate end of data represented as a list of lists of data  */
*tot_output_ptr++ = '\)';
   /* Terminate the retrieval into total_output */
 *tot_output_ptr++ = '\0';
```

The **sqlcl_Execute_Select** function finally returns the number of rows of query data retrieved. Also, the side effect of placing the retrieved query data into *total_output* has occurred.

3.3 *The sqlcl_Print_Row Function*

Data type coercion between information stored in a strongly typed DBMS and untyped Lisp will be examined, via example, in this section. In the code described in Section 3.2, the **sqlcl_Execute_Select** function called the **sqlcl_Print_Row** function to produce the row data. Basically, the DBMS dynamic SQL utility has processed the SQL query and the results have been placed in the SQLDA and SQLCA. The cursor is "pointing" to a row of the database result table. Now the columns in that row of data are handled one by one.

A code fragment of the **sqlcl_Print_Row** function is shown below to illustrate the type of processing involved. An integer stored in column of a row of the database result table is converted into a character representation of the integer, i.e., the character representation of the integer is obtained, and this is what is returned to Lisp, where the Lisp reader can be used to "regenerate" the integer.

In the code below, the sqv pointer to a structure is used. It is declared in the function as

```
IISQLVAR *sqv; /* pointer to 'sqlvar' */
```

IISQLVAR is a typedef for a structure for a single SQLDA (SQL Descriptor Area) variable. A variable is one column (attribute) in a row of the database result table. The type of the variable can be obtained by accessing the *sqltype* field of the structure. The data for the variable is found in the *sqldata* field.

```
/* find the base type of the result (non-nullable) */
if ((base_type = sqv->sqltype) < 0)
        base_type = -base_type;

switch (base_type)
{ case IISQ_INT_TYPE:
   /* All integers were retrieved into long integers.
     Copy the bytes of the long integer one by one into the
     char array that is the same 'dimension' as a long integer */
   for(j=0, p=(char *)sqv->sqldata; j < sizeof(long); j++, p++)
      long_int_copy[j] = *p;
   long_tmp2 = (long *)long_int_copy; /* Caste ptr to copy into ptr to long int */
   long_tmp2_val = *long_tmp2; /* Now get the value of that bugger */
   /* Finally, sprintf it. What a language */
   sprintf_buf_ptr = sprintf(sprintf_buf,"%ld ",long_tmp2_val );
   while (*total_output_ptr++ = *sprintf_buf_ptr++);
   total_output_ptr--; /* back out of the '\0' */
   break;

   case IISQ_FLT_TYPE:
   /* All floats were retrieved into doubles */

        . . .

} /* end switch on base type
```

Notice that the C **sprintf** function is used to produce the character representation of the integer retrieved from the database. All of the processing to copy the bytes of the long integer from the database into the long_tmp2_val variable for **sprintf** are done to maintain word alignment for the RISC processor. Let's examine this in more detail, using integers (as shown in the code fragment), as an example.

When a row of data in the database result table is retrieved from the database, it is placed into a single character buffer in the SQLDA. The bytes of an integer are placed in the character buffer in locations which may not necessarily fall on word boundaries. The bytes of the integer must thus be copied out of the character array a byte at a time before working with the information as an integer. In other words, you can't just access the bytes of an integer at a pointer location in the character array for use as an integer. The integer must be put in a location aligned on a word boundary or a word alignment error occurs.

3.4 Lisp/C Foreign Function Interface

The use of the Lisp/C foreign function interface for SQLCL can best be illustrated by example The **sql-query-db** function described in Section 3.1 calls the **ingres-sqlcl-out-run-query** foreign function defined below.

```
(def-foreign-function (ingres-sqlcl-out-run-query (:name "sqlcl_out_run_query")
                                                  (:language :c)
                                                  (:return-type (:pointer :character)))

   (arg1 :simple-string)
   (arg2 :fixnum) ;when 1, strings are built \"foo bar\" vs. foo bar
)
```

The foreign function definition includes the C component entry point name **sqlcl_out_run_query** which was discussed in Section 3.2. Also, the function definition specifies that a pointer to a character string will be returned. Lastly, the arguments that will be passed to the C function are specified.

4 Issues, Retrospective, and Future

The SQLCL module could be improved. A capability to process database errors in Lisp is needed. A shared structure between the Lisp component and the C component for communicating database error information is one such possibility. Another useful enhancement would be the ability to have multiple databases open simultaneously.

More attention could also be paid to reducing dynamic memory consumption in both the Lisp and C components of SQLCL. In our applications, some of the information in the list of structures produced by **structify-tuple-data** is eventually placed into CLOS [Bobrow et al., 1988, Keene, 1989] instances. This could be very inefficient for applications that require access to large amounts of data from the database. Let's describe why this is true

The database result table is returned to Lisp as a character string, which is then transformed into a list of lists by reading the character string with **read-from-string**. The list of list is then trans-

formed into a list of structures and then slots in the structures are accessed and values are then placed in CLOS instances. All this transformation of the data consumes a lot of CONS cell. This is acceptable for our applications, as our data needs from the database are fairly modest, i.e., the consumption of a few tens of thousands of bytes of extra memory was not a big concern. With heavy database access, though, this could quickly get out of hand.

The Lucid Common Lisp foreign function interface supports the specification of shared structures between a Lisp component and a C component. Thus, a useful enhancement of SQLCL would make use of this facility to place the database result table directly into a set of structures shared between the Lisp and C components, thus eliminating some of the unnecessary CONSing.

There is an enhancement beyond the use of shared structures between the Lisp and C components. It would be nice to be able to define shared storage between C and CLOS instances and have the data placed directly into slots in the shared instances, possibly on a "when needed" basis. Shared instances are not currently supported in the foreign function interface in Lucid Common Lisp. There are many issues surrounding a "when needed" capability. The PCLOS system referenced earlier has investigated several dimensions of this problem.

5 Conclusion

Although we recognize there are improvements that can be made, the current SQLCL module adequately get the job done for us in its current form, i.e., we can utilize a DBMS from our application prototypes. The database is utilized via SQL statements expressed as strings in Lisp. Results are returned to Lisp as a character string. This can be converted into a list of structures, providing a nice functional interface, via the structure accessors, to each attribute (column) in a row of the database result table.

The SQLCL module provides a simple interface between a Common Lisp program and a relational database management system, such as ULTRIX/SQL, that supports dynamic SQL statement processing. More elaborate interfaces to persistent storage from a Lisp application, such as the PCLOS and DOM systems discussed in the paper, could use a system such as SQLCL for some of their database processing.

Acknowledgments

I would like to thank Mark Adler and Rose Horner for reviewing and critiquing this paper and for being supportive and intellectually stimulating co-workers. A thank you is also due to Mike Carifio, who was my inspiring engineering manager when SQLCL was developed, and Steve Schwartz, for reviewing and evaluating my SQLCL development as it progressed. Finally, thanks to the Artificial Intelligence Technology Center and Digital Equipment Corporation for providing a super environment and culture for performing AI research and engineering work.

References

[Bobrow et al., 1988] D. G. Bobrow, et al. "Common Lisp Object System Specification X3J13 Document 88-002R." *SIGPLAN Notices.* Special issue. September 1988 [23].

[Keene, 1989] Sonya Keene. *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Addison-Wesley, Reading, Massachusetts: 1989.

[Manola 1990] Frank Manola. "Object-Oriented Knowledge Bases." *AI Expert.* April, 1990, pp. 46-57.

[Paepcke 1989] Andreas Paepcke. "PCLOS: A Critical Review." In proceedings of the *1989 Object-Oriented Programming: Systems, Languages, and Applications Conference (OOPSLA-89).* 1989, pp. 221-237.

[Sedayao and Irwin 1987] Jeff Sedayao and John Irwin. "CLING - Common LISP/INGRES Interface Manual", Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

[Steele, 1990] Guy L. Steele Jr. *COMMON LISP: The Language.* Second edition. Digital Press, Bedford, Massachusetts. 1990.

[Stonebraker et al., 1976] M Stonebraker, E. Wong, P. Kreps, and G. Held. "The Design and Implementation of INGRES." *ACM Transactions on Database Systems*, 1, 1976, pp. 189-222.