An Open Pre-Compiler for Embedded SQL

Alan Walker
Mike Benzinger

January 5th, 2004

1 Problem Statement

Many commercial RDBMS have a pre-compiler that converts embedded SQL macros into inline code for SQL calls. Each vendor generates calls to their own proprietary, sometimes undocumented, APIs.

We have a large system (>500 KLOC) with approximately 150 modules using embedded SQL. Initially, we wanted to test it against a variety of databases, both commercial and open-source, as part of a port to Linux. Some of the databases we tested, including MySQL, don't have a pre-compiler for ESQL and rewriting these modules to the various database vendors APIs was not feasible.

Initially, we searched the web and also posted a request to several database newsgroups, asking if an open-source pre-compiler was available. We received only a couple of responses, which could be summarized as, "I need one too, please let me know if you find one". So, this seemed like a worthwhile project and something that we should make available to others.

The pre-compiler itself is an awk script and we have a different version for each target database, scripts have been developed for MySQL, ODBC, Oracle and PostgreSQL.

> *Note – our current precompiler has code specific to handling some quirks of NonStop SQL/MP, most notably with date formats. To use this code to port from other database systems will require some minor changes.*

2 Embedded SQL

2.1 Sample Program

The following sample program selects rows from a sample table and prints them. Error handling has not been included, for clarity.

```
#include <stdlib.h>
#include <stdio.h>

/*------------------------------------------------------------------*/
EXEC SQL INCLUDE SQLCA;
short sqlcode;

EXEC SQL BEGIN DECLARE SECTION;
int                                                          host_a;
double                                host_b;
char                                                         host_c;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE csr1 CURSOR FOR
SELECT a, b, c
  FROM table1
 WHERE x = :hostvar1;

/*------------------------------------------------------------------*/
void main (void)
{
                                hostvar1 = 42;

                              EXEC SQL OPEN csr1;
                                if (sqlcode < 0)
                                                                exit(0);

                           while (rc >= 0 && rc != 100)
                                           {
                                                  EXEC SQL FETCH csr1 INTO :
host_a, :host_b, :host_c;
                                               printf("Fetch %d, %lf, %s\n",
host_a, host_b, host_c);
                                           }

                              EXEC SQL CLOSE csr1;
}
```

The pre-compiler approach is used in C/C++, COBOL & PLI, but this article discusses only C/C++. The resulting code is only compilable by a C++ compiler. For each sample statement, we show the code generated for MySQL and for ODBC.

2.2 Declaring Host Variables

```
EXEC SQL BEGIN DECLARE SECTION;
int                                 host_a;
double                              host_b;
char                                host_c;
EXEC SQL END DECLARE SECTION;
```

The database API usually requires the programmer to specify the type of the host variable, which means the pre-compiler generally has to parse the declarations to understand the types. Some databases are limited to simple declarations for host variables, but some pre-compilers allow structures, typedefs, etc in the declare section. This was the case for us and we didn't want to completely parse the C structures.

It occurred to us that we could generate C++ code from the initial C & ESQL code, then write a light-weight wrapper around the database API that uses polymorphism to ensure the correct types are used. The C++ compiler then takes care of the hard parts.

The awk script simply takes the declare section and comments out the EXEC SQL macros, for both MySQL and ODBC interfaces.

```
// EXEC SQL BEGIN DECLARE SECTION;
int                                 host_a;
double                              host_b;
char                                host_c;
// EXEC SQL END DECLARE SECTION;
```

2.3 Declaring a SELECT statement

The following sample shows a typical declaration of an SQL cursor for a select statement:

```
EXEC SQL DECLARE csr1 CURSOR FOR
SELECT a, b, c
  FROM table1
 WHERE x = :hostvar1;
```

The pre-compiler creates a static variable with the text of the SQL statement, along with additional items in the structure to know if the statement has been prepared or not. For MySQL, the following code is generated:

```
// EXEC SQL DECLARE csr1
static e2mysql csr1 = {
    " SELECT a,b,c FROM table1 WHERE x = :hostvar1"
  , NULL
  , 0
};
```

For ODBC, it's almost identical. One modification is to replace the host variables by "?" instead of the names:

```
// EXEC SQL DECLARE csr1
static e2odbc csr1 = {
    " SELECT a,b,c FROM table1 WHERE x = ?"
  , false
  , SQL_NULL_HSTMT
};
```

2.4 Executing the SELECT statement

The SELECT statement is executed when the program opens the ESQL cursor:

```
EXEC SQL OPEN csr1;
```

The first step in executing the statement is to ensure that we're connected to the database. For some systems, with precompiled and bound SQL[1], connections are implicit.

For MySQL, the following code is inserted:

```
// EXEC SQL OPEN csr1
static int16
open_csr1()
{
  try
  {
    if ( ! connectionMade )
      SQLHelperConnect();
    if ( csr1.rslt != NULL )
      mysql_free_result(csr1.rslt);
    SQLBindParmPoly(sqlStmt, ":hostvar1", hostvar1, sizeof(hostvar1));
    if ( mysql_real_query(&mysqlConnection, sqlStmt.c_str(),
                          sqlStmt.length()) )
      handle_error("mysql_real_query");
    csr1.rslt  = mysql_store_result(&mysqlConnection);
    if ( ! csr1.rslt )
      if ( mysql_field_count(&mysqlConnection) != 0 )
        handle_error("mysql_store_result - no results");
    csr1.row  = 0;
    sqlcode = SQL_SUCCESS;
  }
  catch(...)
  {
    sqlcode = SQL_ERROR;
  }
  return sqlcode;
}
//--------------
```

[1] Mainframe DB2 and Tandem SQL/MP are examples of implicit connections

For ODBC, if the statement has never been executed, we first prepare it and bind the host variables used as parameters. The following code shows the generated code:

```
                                // EXEC SQL OPEN csr1
static int16
open_csr1()
{
  try
  {
    if (csr1.stmt == SQL_NULL_HSTMT)
    {
      if ( odbcConnection == 0 )
        SQLHelperConnect();
      sqlcode = SQLAllocStmt(odbcConnection, &csr1.stmt);
      if ( sqlcode == SQL_ERROR || sqlcode == SQL_INVALID_HANDLE )
        handle_error(odbcEnvironment, odbcConnection,
                     csr1.stmt, sqlcode, "SQLAllocStmt");
      sqlcode = SQLPrepare(csr1.stmt, (SQLCHAR*)
                           sqlStatementText.c_str(), SQL_NTS);
      if ( sqlcode != SQL_SUCCESS && sqlcode != SQL_SUCCESS_WITH_INFO )
        handle_error(odbcEnvironment, odbcConnection,
                     csr1.stmt, sqlcode, "SQLPrepare");
      SQLBindParmPoly(csr1.stmt, 1,hostvar1, sizeof(hostvar1));
    }
    sqlcode = SQLExecute(csr1.stmt);
    if ( sqlcode != SQL_SUCCESS && sqlcode != SQL_SUCCESS_WITH_INFO )
      handle_error(odbcEnvironment, odbcConnection,
                   csr1.stmt, sqlcode, "SQLExecute");
  }
  catch(...)
  {
  }
  return sqlcode;
}
//--------------
```

2.5 Fetching Data

The FETCH statement specifies the target variables for the data:

```
EXEC SQL FETCH csr1 INTO :host_a, :host_b, :host_c;
```

When we fetch from the cursor, note that we don't need the specify the types of the variables as we've used polymorphism to let the C++ compiler determine the correct function to call:

```
// EXEC SQL FETCH csr1
static int16
fetch_csr1()
{
  if ( ! csr1.rslt )
    return SQL_ERROR;
  if ( csr1.row >= mysql_num_rows(csr1.rslt) )
    return SQL_NO_DATA;
  MYSQL_ROW row = mysql_fetch_row(csr1.rslt);
  SQLBindColPoly(row[0], host_a, sizeof(host_a));
  SQLBindColPoly(row[1], host_b, sizeof(host_b));
  SQLBindColPoly(row[2], host_c, sizeof(host_c));
  ++csr1.row;
  return SQL_SUCCESS;
}
//---------------
```

For ODBC, similar code is generated:

```
// EXEC SQL FETCH csr1
static int16
fetch_csr1()
{
  try
  {
    if ( ! csr1.bind )
    {
      SQLBindColPoly(csr1.stmt, 1, host_a, sizeof(host_a), &dummy);
      SQLBindColPoly(csr1.stmt, 2, host_b, sizeof(host_b), &dummy);
      SQLBindColPoly(csr1.stmt, 3, host_c, sizeof(host_c), &dummy);
      csr1.bind = true;
    }
    sqlcode = SQLFetch(csr1.stmt);
    if ( sqlcode == SQL_ERROR || sqlcode == SQL_INVALID_HANDLE )
      handle_error(odbcEnvironment, odbcConnection, csr1.stmt, sqlcode,
"SQLFetch");
  }
  catch(...)
  {
  }
  return sqlcode;
}
//---------------
```

2.6 Closing the SELECT statement

Closing the cursor is the simplest step:

```
EXEC SQL CLOSE csr1;
```

This translates to the following code for MySQL:

```
// EXEC SQL CLOSE csr1
static int16
close_csr1()
{
  mysql_free_result(csr1.rslt);
  csr1.rslt = NULL;
  csr1.row  = 0;
  return SQL_SUCCESS;
}
```

For ODBC, we generate the following code:

```
// EXEC SQL CLOSE csr1
static int16
close_csr1()
{
  try
  {
    sqlcode = SQLFreeStmt(csr1.stmt, SQL_CLOSE);
    if ( sqlcode == SQL_ERROR || sqlcode == SQL_INVALID_HANDLE )
      handle_error(odbcEnvironment, odbcConnection, csr1.stmt, sqlcode,
"SQLFreeStmt");
  }
  catch(...)
  {
  }
  return sqlcode;
}
```

2.7 Modified Code

The EXEC SQL statements are now replaced by calls to the generated functions:

```
int main (void)
{
                                hostvar1 = 42;

                             /* Open the cursor */
                  sqlcode = open_csr1();  // EXEC SQL OPEN csr1;
                                if (sqlcode < 0)
                                        {
                                                printf("OPEN fareCursor,
sqlcode = %d\n", sqlcode);
                                        }

                           while (rc >= 0 && rc != 100)
                                        {
                                                sqlcode = fetch_csr1();  //
EXEC SQL FETCH csr1 INTO :host_a, :host_b, :host_c;
                                             printf("Fetch %d, %lf, %s\n",
host_a, host_b, host_c);
                                        }

                  sqlcode = close_csr1();  // EXEC SQL CLOSE csr1;

                                     return 0;
}
```

2.8 Insert / Update / Delete

Handling insert / update / delete statements is relatively straightforward.

The SELECT … INTO statement, which is a single-row read, is also handled as a special case.

3 Implementation

3.1 The awk Code

The pre-compiler is implemented in awk, we typically use GNU awk (gawk) to execute the code. The complete code is less than 1000 lines, including comments.

The code relies extensively on regular expressions to find the embedded SQL in the program, as well as some manipulation of the SQL statements themselves. For example, recognizing the ESQL macros is done with this pattern:

```
toupper($1)=="EXEC" && toupper($2)=="SQL" {
…
}
```

The getStatement() function simply reads through the input file until it reaches the terminating semicolon of the SQL statement. It understands single and double quotes inside the SQL but, like any compiler, will get really confused by unbalanced quotes. This function returns the entire SQL statement as a single string.

There are a few routines for manipulating the SQL statements. We have routines to convert the code to a plain vanilla SQL, such as stripping out Tandem SQL/MP specific syntax. For ODBC, host variables are recognized and substituted by "?". Host variables are recognized with regular expressions:

```
match(sql, /\:[[:alpha:]][[:alnum:]_\.]*(\[[^\]]*\])*/);
```

Most of the awk code generates the required inline code for the database calls. We also have additional options to generate debug code.

3.2 Helper Routines

To get the C++ compiler to match the types, we simply wrap the API with inline functions. For MySQL, we substitute the parameters directly into the SQL statement.

```
inline int32
SQLBindParmPoly(std::string& stmt, const char* text,
                const int32 parm, uint16 /*size*/)
{
  std::string::size_type pos = stmt.find(text);
  if ( pos != std::string::npos )
  {
    char buffer[12];
    snprintf(buffer, 12, "%d", parm);
    stmt.replace(pos, strlen(text), buffer);
  }
  return SQL_SUCCESS;
}
```

Returned values for MySQL:

```
inline int32
SQLBindColPoly(const char* value, int32& parm, uint16 /*size*/)
{
  parm = atoi(value);
  return SQL_SUCCESS;
}
```

For ODBC query parameters:

```
inline int32
SQLBindParmPoly(SQLHSTMT hstmt, SQLUSMALLINT col, const int32&
parmPtr, SQLINTEGER bufLen)
{
  int32 rc = SQLBindParameter(hstmt, col, SQL_PARAM_INPUT,
                              SQL_C_SLONG, SQL_INTEGER, bufLen, 0,
                             (void*) &parmPtr, bufLen, 0);
  if ( rc == SQL_ERROR || rc == SQL_INVALID_HANDLE )
    handle_error(odbcEnvironment, odbcConnection,
                 hstmt, rc, "SQLBindParameter:int32");
  return rc;
}
```

The downside to using polymorphism is that some older ESQL / C programs might rely on deprecated K&R C compilers, causing problems when converted directly to C++. In our case, we didn't run into these problems and we felt it would be better to update the code if we did find such problems.

3.3 Some More Details

The pre-compiler handled some minor SQL differences from one dialect to the next. HP's NonStop SQL requires dates to be handled in a unique fashion.

The pre-compiler uses #line directives so that programmers can debug against the original source code they wrote. It also adds the necessary include directives for the helper routines.

As an additional touch, it attempts to follow the same indentation pattern as the original code as the authors believe that generated code doesn't have to be ugly or unreadable.

3.4 To-Do List

1. The current code is not particularly generic, in that we have a separate awk script for each target database. There is a lot of duplication between the scripts.
2. We've discussed rewriting it in perl, as it may get more traction in the open source community than an awk script.

3. Remove / isolate some of the code that's specific to our application.
4. A more generic pre-compiler may need some more robust pattern matching. We prefer, however, to avoid creating a yacc grammar as minor syntax differences between various vendors' proprietary ESQL pre-compilers could make our pre-compiler fragile.

4 Conclusion

The code described in this paper is used every day to compile a large production system, running on Linux IA-64, using MySQL. During our benchmarking efforts, we already had the pre-compiler running for ODBC, PostgreSQL and Oracle when we came to the MySQL port. Overall, it took 2 days to get our entire system running on MySQL, using ESQL / C. This was much easier than we expected.