Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Debugging User-Defined Functions in RDBMS Client-Server Environment*

Gene Y. Fuh[1] *Kevin Nomura*[2] *Mike Meier*[3] *Hsin Pan*[3] *George Wilson*[1]

International Business Machines Corporation
Email: fuh@almaden.ibm.com
Phone: (408) 927-3113 Fax: (408) 927-3215

Abstract

Many relational database systems offer a User-Defined Function (UDF) facility which allows the user to extend the intrinsic functionality of SQL expressions with their own functions written in languages such as C and C++. UDF modules are installed on the database server where they run under a daemon process managed by the DBMS. Normal debugging techniques fail with UDF in the server runtime environment because client cannot readily communicate with UDF, which runs as an asynchronous, remote, privileged process from the client's point of view.

Developing and testing non-trivial UDF modules may not be viable without a ready means to debug in this environment. In this paper we explore the factors that inhibit UDF debugging. We then present an idea that overcomes these obstacles and develop two variations of this idea. The self-initiated *approach can be inexpensively applied by modification of the UDF code. We demonstrate a simple implementation of this technique which was experimented with using IBM's DB2/Common Server relational database. The* DBMS-initiated *approach has DBMS managing the debugger invocation, with control provided through SQL language extensions, to make debugging more convenient and flexible for the developer of UDF. Both approaches allow normal debugging of UDF in client-server environment using ordinary debuggers.*

Keywords: relational database management system (RDBMS), SQL language, SQL extension, debugging, user-defined function (UDF), client-server, stored procedure, parallel query evaluation.

* [1]IBM Database Technology Institute [2]Apple Computer Inc. [3]IBM Application Development Technology Institute

1 Introduction

Many relational database management systems (RDBMS) [18, 10, 19] currently offer a User-Defined Function (UDF) [15] facility whereby the user may extend the intrinsic functionality of SQL expressions with their own functions written in host languages such as C and C++. A UDF can be invoked from any context where an SQL expression [14] is allowed. Thus UDF provides a flexible mechanism for integrating databases with applications. Some uses of UDF are accessing multimedia objects through multimedia extenders [6, 9], presenting relational data through Web gateways [17], supporting data mining applications [2, 1], and wrappering system library functions [8] so they can be called from SQL.

Should the UDF logic contain errors, the SQL expression in which the UDF was invoked may give incorrect results or fail. This of course is a form of user error since UDF semantics are specified by the developer writing UDF code not by the DBMS or the SQL language. Although the UDF developer has the usual responsibility for correctness, debugging UDFs running in the client-server environment is severely hampered by the characteristics of that environment. We will discuss these problems and propose a solution which makes UDF debugging as effective as normal client application debugging.

We now describe a representative UDF run-time environment [7] to illustrate the special problem of debugging remote, DBMS-managed UDFs. The diagram of figure 1 depicts the elements of DBMS which directly relate to UDF debugging.

Each client connecting to the DBMS has a set of server processes associated with it. Such a set is shown as *Agent unit* in the system diagram (figure 1). To use an already registered *UDF*, the client requests its *agent process* to execute an SQL expression which

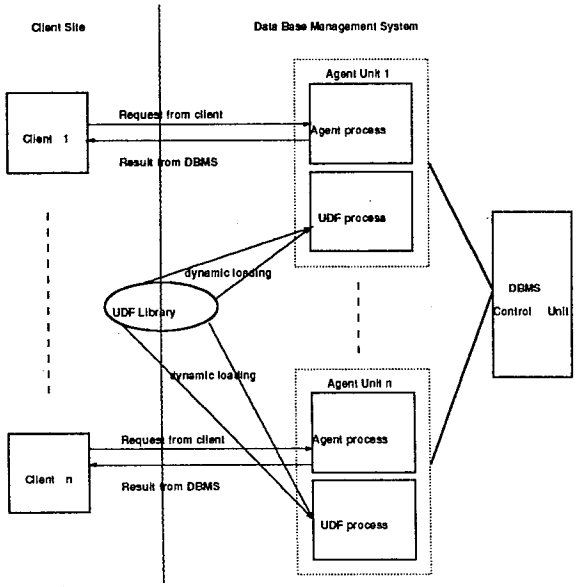

Figure 1: Run-time environment for User-Defined Function

contains a *UDF* invocation. In response, *agent process* interprets the expression and returns the result to the client.

When a UDF is installed in the server environment, the user gives up direct control over its loading and execution; these are now managed by DBMS. Viewed from the client side of a transaction, the UDF appears as an asynchronous, remote, privileged process. This makes it rather difficult to target a debugger to the UDF from the client side. The three characteristics of special interest in this runtime environment are:

Timing: A *UDF* context process is created on demand and the *UDF* library is loaded dynamically. Unloading occurs at the end of a transaction, such as when a `COMMIT` is performed. There is no convenient way to inform the developer when *UDF* code is available for debugging. The debugger cannot be attached to *UDF* on the fly as process id and debugging information of *UDF* library cannot be known in advance.

Authorization: For security purposes, DBMS processes usually run under a special UID so that non-privileged users cannot attach to them.

Remote debugging: In general the developer does not have a user account on the server machine. This makes it extremely difficult, if not impossible, to debug *UDF process* on the server. Furthermore, *UDF process* runs as a daemon process with the standard I/O file handles disabled. Therefore, run-time status of UDFs can not be made observable by inserting *printf* statements.

Normal debugging methods involving interactive debuggers and even "print" statements added to the program cannot be used with these limitations. The developer, working on a remote machine, might find it frustrating to debug a misbehaving UDF. The reloading of *UDF* libraries is one example of a factor of the runtime environment that could potentially confuse the user. The user may wonder when his code is being refreshed, why static variables are being reinitialized, etc. This could affect the productivity of application development using *UDF* and even discourage users from using *UDF*.

Relying entirely on debugging the UDF off-line, i.e. on the developer's machine outside the DBMS context, is too limited in general. Many factors of the environment and function inputs which trigger the failure may be unreproducible. In general a *UDF* can use any feature of the host language and be arbitrarily complex, so it is very desirable that familiar debugging methodologies be available to the developer. In particular we believe the ability to easily debug a *UDF* running online, that is in the environment of the DBMS, is central to the viability of writing non-trivial UDF modules. Online debugging support complements the standalone development and testing of *UDF* code by exposing behavior unique to the *UDF* runtime environment and making it convenient to reproduce the situation causing the failure. The ability to debug UDF problems *in situ* will be invaluable to the application developer.

The remainder of this paper develops our approach to UDF debugging in the client-server environment. In section 2, we present a technique for debugging *UDF* that resolves the problems raised above. Section 3 demonstrates a simple implementation of this idea along with an example of its use. This idea has been prototyped in IBM DB2/CS Version 2 [8] where its usefulness was verified. In section 4, we refine the idea by proposing DBMS support for UDF debugging. We conclude in section 5.

2 Our approach

Summarizing the previous discussion, there are three obstacles to debugging DBMS-managed UDF code from the client side. These are the issues of **timing**, **authorization** and **remote debugging**.

These issues may appear to be orthogonal at first glance. However it turns out they are caused by transplanting the same methodology for developing and de-

bugging the UDF locally to debugging the UDF when installed on the server and managed by DBMS. If we try to initiate the debugger from the client side we are then faced with having to target the debugger to an asynchronous, remote, privileged process. But the UDF itself is immune to these problems in an important sense, which leads us to a solution: *having UDF initiate the debugger.*

The timing problem, of trying to intercept the UDF invocation under DBMS control, lies in the perception of the UDF process as observed by another process. Relative to the UDF this is not a problem; *UDF process* can initiate the debugger right before control passes to the UDF code, providing the same measure of control as local debugging. Likewise, insufficient authorization to attach *UDF process* is an artifact of the relationship between UDF and client processes. If *UDF process* initiates the debugger, it only requires permission to attach itself.

To debug the remote process, we observe that a login account on the server machine is not needed to debug a server process, as long as we can have the debugger interface brought up elsewhere: We can use the fact that UDF already runs on the server and have UDF initiate a debugger in client-server mode, with the debugger interface brought up on the application side. For example, X-windows is sufficient for flexible remote debugging of UDF. We can run a text debugger (gdb, dbx) by means of an xterm window with options to display on the application's X-server and to run the debugger. Using a graphical debugger is equally straightforward.

Therefore, the **timing**, **authorization** and **remote debugging** issues reduce to the problem of initiating a debugger from *UDF process* for debugging *UDF* code running in its own address space. There are two approaches to achieving this: *self-initiating* approach and *DBMS-initiating* approach We will illustrate the basic idea of both approaches with a segment of C code which is executed right before the execution of *UDF* code. Without loss of generality, we assume the underlying symbolic debugger is "xldb" [4], an IBM symbolic debugger.

In the *self-initiating* approach, the debugger is "triggered" by code added to the UDF for this purpose. The following section of code illustrates the basic mechanism:

```
{
  char debug_cmd[256];

  sprintf(debug_cmd,
          "xldb -a %d -I %s -display %s %s &",
          getpid(), source_path,
          display, program_name);
  system(debug_cmd);
  sleep(5);   // synchronize w/ spawned debugger process
}
```

The first statement prepares a shell command for invoking the "xldb" debugger. The "-a" option specifies the process id of the process being debugged. The "-I" option specifies the directory where the source file is located. The "-display" option specifies the remote machine and X-windows display that the developer is using. The last argument on the "xldb" command line is the name of the program to be debugged.

Notice that this command is run as a background job so that the UDF process does not have to wait for its completion. Execution of the `system` call spawns the debugger in a new process running the prepared command; meanwhile UDF execution continues.

The `sleep` call gives the debugger time to attach back to the UDF process before UDF execution continues. UDF execution will break within the `sleep` function to give the overall effect of establishing the initial breakpoint within this section of code.

Alternatively, in the *DBMS-initiating* approach, the debugger is "triggered" from within the DBMS. The mechanism of debugger attachment is similar to the above. But, having DBMS manage the debugger invocation provides more flexibility to the developer, as we shall see later in section 4.

In both approaches, the *UDF* of interest can be debugged in the usual way. We will describe a simple implementation of this idea in the following section.

3 Implementation and example

We have implemented the *self-initiating* [5] approach using both "xldb" and "dbx" as the underlying symbolic debugger in the context of IBM DB2/Common Server version 2 running under AIX/6000. To simplify the debugging process, we provide a C macro, `DEBUG_UDF`, for application programmers to specify minimum set of debugging information. The definition of `DEBUG_UDF` is as follows:

```
#define DEBUG_UDF(dbg, src_dirs, display, ulst,
                  uarg, flag)
{
  static int token;
  token = dbf_register(dbg, src_dirs, display,
                       ulst, uarg);
  if (token >= 0)
     flag = dbf_trigger(token);
}
```

Argument *dbg* specifies the name of the debugger executable ("dbx", "xldb", etc). Argument *src_dirs* specifies a colon-separated list of *UDF* source directory names. Argument *display* specifies the user's X display as *machine:display-number*. Argument *ulst* specifies a colon-separated list of *UDF* names. Argument *uarg* is the formal argument of the enclosing C function which contains the full function name of the

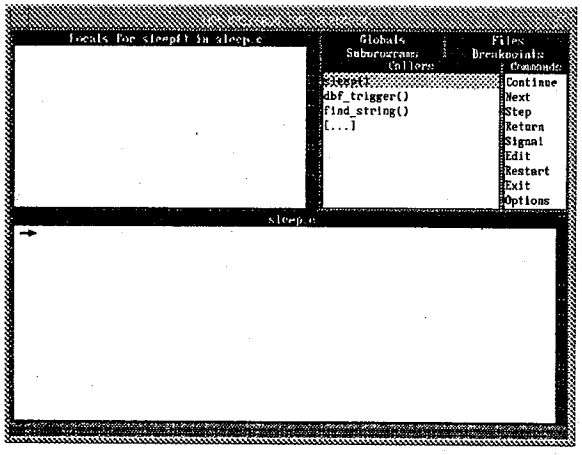

Figure 2: xldb *screen when debugger instantiated*

UDF being executed. Argument *flag* is a local "int" variable name for holding the return code from the "trigger" routine whose sole functionality is to bring up the debugger.

It takes two statements to initiate the debugger.

The first statement registers the current debugging session with the *DBMS*. If the name specified by *uarg* is not in any of the existing registered debugging sessions, *dbf_register* will create a new session and registers the debugging information as specified by the arguments. A token uniquely identifying this session will be returned as the result. If the name specified by *uarg* is found in some existing session, token identifying that session will be returned as the result.

The second statement attempts to bring up the debugger if the current process is not being traced and no attempt was made before to bring up the debugger.

The following code fragment shows the use of this macro to specify the initial breakpoint at entry to the UDF:

```
DEBUG_UDF
  ("xldb",                          // IBM xldb debugger
   "/afs/alm/u/nomura/udf/lib",     // debugger src search path
   "bughouse.almaden.ibm.com:0",    // X-svr for debugging
   "nomura   .FIND_STRING",         // DBMS identifier for UDF
   udf_func_name,                   // parameter passed from DBMS
   flag);                           // output parameter
if (flag != 0)
   /* debugger was not brought up successfully */
   ...
else
   /* debugger was brought up successfully */
   ...
/* Start of function body */
```

"db2dbf.h" is a header file containing the definition of DEBUG_UDF and the function prototypes for dbf_register and dbf_trigger. The DEBUG_UDF line in the function body of find_string registers a new debugging session with the following debugging information:

- **Debugger:** IBM xldb debugger.
- **Source directories:** AFS directory '/afs/alm/u/nomura/udf/lib'.
- **Client machine:** X-windows server "bughouse.almaden.ibm.com:0".
- **UDF name:** The *UDF* FIND_STRING in schema NOMURA.

If all of this information is correct and "NOMURA .FIND_STRING" is invoked the *xldb* debugger will be brought up and break at the DEBUG_UDF line. As desired, the *xldb* window will be displayed on "bughouse.almaden.ibm.com".

We would like to comment more on the *uarg* and *ulst* arguments of DEBUG_UDF. At run-time, *uarg* will point to a string which contains the complete two-part name of the *UDF* being invoked. If there are multiple *UDF*s sharing the same implementation this argument identifies the one which is being invoked. Suppose there is another *UDF* "FUH .FIND_STRING" which is also implemented by the same external function. With the *ulst* argument given in the above example, the DEBUG_UDF line will have no effect if the *UDF* being invoked is "FUH .FIND_STRING".

To summarize the usage of our debugging support, the *UDF* should be prepared as follows. Then once the library is installed, the next invocation of the *UDF* will initiate the selected debugger.

- Include the header file "db2dbf.h" in *UDF* source file.
- Add the DEBUG_UDF line as explained above to the *UDF* source file where the initial breakpoint is desired.
- Recompile with debug support activated (i.e., specify the -g option).
- Link *UDF* library with the shared library "libdbf.a" which contains the functions *dbf_register* and *dbf_trigger*.

3.1 UDF debugging example

We now show an example of debugging an error in UDF using this technique. Following is the complete definition of a C-language UDF `find_string`, whose purpose is to return the position in *text* of the first occurrence of *sub_string*. If *sub_string* does not occur in *text*, the return value is 0. At run-time the DBMS prepares and passes actual arguments to external C

function `find_string` via its first two parameters, *text* and *sub_string*. The return value is passed back to the DBMS via the third parameter, *position*, and the sixth parameter *null_ind_r* which indicates when the result is null. The other parameters of `find_string` are unrelated to our discussion.

Here is the UDF source code:

```
void find_string
 (char  *text,          // The text to search
  char  *sub_string,    // The sub-str
  long  *position,      // Occurring pos. of the sub_str
  short *null_ind_i1,   // First input null-indicator
  short *null_ind_i2,   // Second input null-indicator
  short *null_ind_r,    // Result null-indicator
  char   sqlstate[6],   // Error code issued by UDF
  char  *func_name,     // UDF function name
  char  *specific_name, // UDF specific name
  char  *msg_text)      // Msg txt returned by UDF
{
  char *start, *cptr1, *cptr2;
  int   start_position;

  /* Initialization */
  start = text;
  start_position = 1;

  /* Set up new string for comparison */
try_again:
  *position = start_position++;
  cptr1 = start++;
  cptr2 = sub_string;

  /* Compare sub_string with the new string  */
  /* The first two if-branches should be swapped */
compare:
  if (*cptr1 == '\0') {
     /* End of text reached; not found */
     *position = 0;
     return;
  }
  else if (*cptr2 == '\0') {
     /* End of sub_string reached; found */
     return;
  }
  else if (*cptr1 == *cptr2) {
     /* So far so good; continue comparing */
     cptr1++;
     cptr2++;
     goto compare;
  }
  /* Comparison fails; try next string */
  goto try_again;
}
```

The C code given above contains a couple of defects: first, the null result indicator is not initialized, which may cause the function to appear to return NULL; second, it does not catch the situation where the sub-string ends the text. Let *test_table* be defined with a single column *text* of type *LONG VARCHAR* loaded with three rows:

```
TEXT
------------------------------
this is a test
this is only a test
if this had been a real emergency it would not have worked
```

The *UDF* is now prepared for debugging. We choose to place a breakpoint at the top of the function as illustrated by the insertion of `DEBUG_UDF` in the fragment of code in the previous section. After recompiling and installing the library, we are ready to begin a debugging session. We run an interactive query from the command-line processor formulated to show the problem by searching for a string that occurs at the end of the first *text* row:

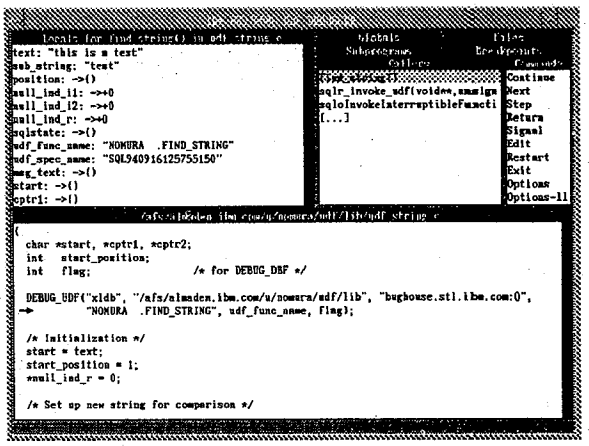

Figure 3: *Ready to debug*

```
select text from test_table
            where find_string(text, 'test') <> 0
```

Presently an *xldb* debugger window appears (figure 2). The application (command-line processor in this case) remains suspended waiting for a response from the agent process to the SELECT query while we work in the debugger.

The debugger is stopped two call levels below the `DEBUG_UDF` line in a `sleep` call that is used to suspend the *UDF* process long enough for the debugger to spawn off and attach. The two call levels expose the details of the mechanism for invoking the debugger which are irrelevant to the *UDF*. By issuing two *return* commands to the debugger, we reach the "logical" breakpoint as shown in figure 3.

Local variables are shown in the upper left window pane. The arguments passed to the *UDF* from the DBMS appear at the top; for example `text` points to the first row value and `sub_string` points to the value to search for, as specified in the *UDF* invocation. At this point *UDF* debugging can proceed as normal.

Since `find_string` incorrectly returns that "test" does not occur within "this is a test", we set a breakpoint on the condition which decides that a substring is not found. The breakpoint hits as shown in figure 4. Examining the local variables, we see that everything is actually as we would expect, having exhausted both search and target strings, except we are about to return that the string is not found. This leads us to realize that the order of the first two `if` conditions is reversed, since this state should be considered a substring match. We are finished debugging for now. To end debugging, the debugger must be terminated gracefully so the process state is restored to a runnable condition (closing the xldb window with ALT-F4 for example would terminate the UDF process, and a se-

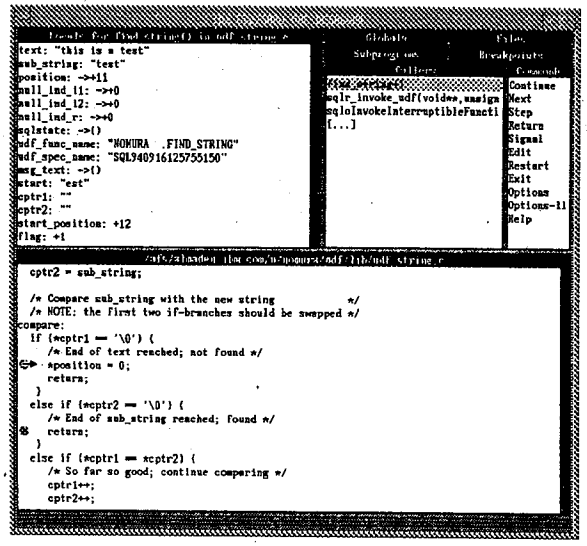

Figure 4: *Bug discovered*

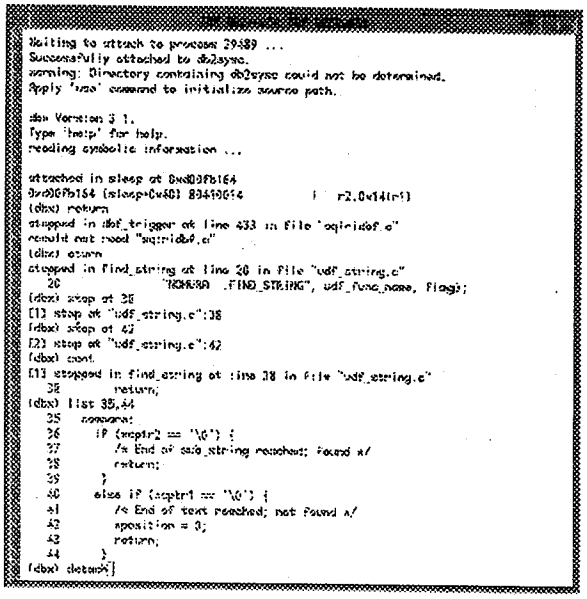

Figure 5: *Bug fixed*

vere error would be reflected to the application). Selecting the "xldb" exit option removes all breakpoints and closes the debug session. Control now returns to the application (command-line processor), and the query we issued resumes running.

To fix the UDF we terminate the command-line processor so as to end the current unit of work, or alternatively perform a COMMIT. This is a necessary step to cause the *UDF* library to be reloaded upon the next reference instead of using the copy already loaded into memory. We update the code and install the library, and we are ready to begin again.

This time for variety the "dbx" flavor of *UDF* debug support is used. We set a breakpoint at the "found" and "not found" return points and run the function to verify the correct path is taken this time. As shown in figure 5 this bug has been fixed.

After releasing the UDF process from debugger control with the detach command we observe that the query output is now correct; exactly those rows containing "test" are returned.

```
select text from test_table
          where find_string(text, 'test') <> 0

TEXT
------------------------------
this is a test
this is not a test

2 record(s) selected.
```

4 DBMS extensions for UDF debugging

More advantages can be obtained by integrating our debugging technique into the underlying DBMS. First, the intent of debugging specific UDFs can be expressed to DBMS through an extension to existing SQL languages. It is then unnecessary to modify UDF source code with statements to initiate debugging, as DBMS manages this task. Conditional breakpoint parameters can be registered with the DBMS so that the debugger is invoked for a given UDF only when certain conditions in the encapsulating SQL statement are met. As examples, one can specify that the target UDF is debugged only when the first argument of the UDF is a *date* value greater than the current *time stamp* or it is the last time that the UDF is invoked from the underlying SQL statement. Second, existing database authorization mechanisms can be extended to include the debugging operation over UDFs. In other words, only users granted with appropriate *debugging privileges* are allowed to debug the UDFs. In this section, we propose an extension to the existing SQL language and provide a variation of the technique introduced in section 2.

4.1 SQL Language Extensions

The following grammar rules defines the syntax of the *set_debug_intent* statement that we propose as an extension to the existing SQL language:

```
set_debug_intent_stmt ::=
  SET DEBUG INTENT FOR udf_list WITH debug_intent_list

debug_intent ::=
    CONTROL = {ON | OFF}
  | DEBUGGER = string
  | OPTIONS  = string
  | DISPLAY  = string
```

```
| CONDITION = debugging_condition
```

The *udf_list* in the first rule specifies the list of UDFs to which the debugging intents specified in the statement apply. There are five kinds of intent. The *CONTROL* intent specifies whether the underlying debugger should be activated when the desired UDF invocation is encountered. The *DEBUGGER* intent and the *OPTIONS* intent specify the name of the debugger and the command line options for the execution of the debugger respectively. As suggested by its name, the *DISPLAY* intent defines the location where the output of the debugger is rendered. The *CONDITION* intent, specifies a boolean condition under which the debugger is expected to be activated. The condition is an SQL expression in which the tokens #1, #2, etc. can be used to denote parameters of the UDF. All the debug intents are optional. The default values are not defined and hence are implementation-dependent.

The usage of the *set_debug_intent* statement can be best understood through examples.

```
SET DEBUG INTENT FOR find_string WITH
    CONTROL = ON,
    DEBUGGER = 'xldb',
    OPTIONS = '-I /afs/almaden.ibm.com/u/fuh/udf/src',
    DISPLAY = 'ingrid.almaden.ibm.com:0',
    CONDITION = LENGTH(#1) > 0 AND LENGTH(#2) > 0;
```

The above statement instructs the DBMS to spawn a new process running the *xldb* debugger on behalf of the UDF *find_string* when it is invoked with non-empty string-typed values for its two arguments. The debugger window will be rendered on the machine whose IP name is "ingrid.almaden.ibm.com". The following statement will deactivate the debugger for the UDF *find_string*. As a result, subsequent invocations of *find_string* will run without interruption by the debugger.

```
SET DEBUG INTENT FOR find_string WITH CONTROL = OFF;
```

4.2 Activation of the Debugger

The DBMS uses a variation of the technique described in section 2 to attach a debugger to *UDF process*. In the DBMS-initiating mode, the debugger is "triggered" by the DBMS prior to the invocation of the UDF. The following section of code illustrates the basic mechanism:

```
{
  char  debug_cmd[256];
  sprintf(debug_cmd,
          "xldb -a %d -I %s -r %s -display %s %s &",
          getpid(), source_path, function_name,
          display, program_name);
  system(debug_cmd);
  sleep(5);
}
```

As with the self-initiating technique described in section 2, this prepares a shell command for running the debugger as a separate process. The difference here is the program_name final argument which specifies the name of the program to be debugged.

Executing the system call spawns a new process running the prepared command and returns to the DBMS immediately. The sleep call keeps the running process idle for a period of time so that the debugger can attach to it before the calling process proceeds to execute the external function. As the result of executing this section of code in UDF process, a new "xldb" debugger process will be brought up and the execution will break at the first executable statement of the external function being debugged.

5 Conclusion

Our experience with a simple implementation of remote debugger instantiation has shown a breakthrough in debugging UDF. With a small amount of coding overhead we easily debugged a UDF under the DB2/Common Server run-time environment on AIX using existing source debuggers. We have also described how this technique can be integrated into existing relational database management systems to enhance the usability of the basic debugging idea. A UDF can be targeted or withdrawn from debugging via SQL commands without source modification, and the DBMS can provide conditional entry breakpoints to simplify reproducing the failure scenario.

5.1 Future works

We point out a couple of areas where the UDF debugging idea can be applied to related problems.

Stored procedures execute on the database server in a runtime environment similar to that of UDF. Therefore stored procedures can be debugged using the *self-initiated* method. With DBMS and SQL extensions analogous to those proposed for UDF, we can provide the advantages of *DBMS-initiated* method for stored procedures.

Debugging UDF in context of a parallel database engine [3, 16] is another challenge. The Parallel and Distributed Debugging Analyzer (PDDA) [13, 12, 11] developed at IBM is a system well suited for such a scenario. For example, PDDA can present a unified session to the user for multiple instantiations of the same UDF code. It can also manage call frames spanning the application code on the client and the UDF or stored procedure code on the server. PDDA solves the problem of remote debugging but the issues of timing and authorization for UDF remain. The

role of our UDF debugging technique is to spawn debugger's agent process which will communicate with PDDA and acquire control of UDF execution, using essentially the same model.

Finally, the remote debugging solution we have presented relies on underlying support for client-server debugging, which is commonly provided by X-windows on UNIX-based servers. On non-UNIX server hosts it is an additional task to develop the remote debugging support.

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. In *IEEE Transaction on Knowledge and Data Engineering*, pages 5(6):914–925, 1993.

[2] Rakesh Agrawal and Kyuseok Shim. Tightly-Coupled Integration of Application Programs with Relational Database Systems: Methodology and Experience. IBM Almaden Research Center Report, 1995.

[3] C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. Db2 parallel edition. In *IBM SYSTEMS JOURNAL, Vol 34, No 2*, pages 292–322, 1995.

[4] Tim Bell. *XFDB – A Symbolic Debugger for AIX Version 2.23 for RISC System/6000.* IBM High Energy Physics European Centre, July 1993.

[5] Gene Fuh. Debugging User-Defined Functions - User Guide. September 1994.

[6] Tri Ha and S. Terry Donn. DB2 V2 Image Extender Specification. FPFS of IBM MMDB, January 1995.

[7] IBM Corporation. DATABASE 2 AIX/6000 and DATABASE 2 OS/2 SQL REFERENCE. First Edition, October 1993.

[8] IBM Corporation. DATABASE 2 SQL REFERENCE for COMMON SERVER - VERSION 2. July 1995.

[9] IBM Corporation. *DATABASE 2 Image, Audio, and Video Extenders: Administration and Programming*, June 1996. First Edition.

[10] Informix. *The INFORMIX Guide to SQL: Syntax, V.6.0*, March 1994. Part 000-7597.

[11] Michale S. Meier, Hsin Pan, and Gene Y. Fuh. Debugging DB2/CS client-server applications. *IBM Systems Journal*, 36(1), January 1997.

[12] Mike Meier, Bob Harding, Len Lyon, Hsin Pan, and Leslie Scarborough. pdda – the Parallel and Distributed Dynamic Analyzer, users guide. Technical Report ADTI-1994-001, Application Development Technology Institute (ADTI), IBM Software Solutions Division, San Jose, California, May 1994. (also available as IBM Technical Report STL TR 03.575.).

[13] Mike Meier, Hsin Pan, Bob Harding, Len Lyon, and Leslie Scarborough. Parallel and Distributed Dynamic Analyzer (PDDA) – a debugger for client/server programs. In *Proceedings of the IBM 1994 Program Technology Forum*, pages 57–76, Yorktown Heights, New York, June 6–7 1994. (An extended and revised version is released as IBM Technical Report ADTI-1994-003 (STL TR 03.571), July 1994.).

[14] Jim Melton, editor. *(ISO-ANSI Working Draft) Database Language (SQL2).* International Organization for Standardization and American National Standards Institute, 1992.

[15] Jim Melton, editor. *(ISO-ANSI Working Draft) Database Language (SQL3).* International Organization for Standardization and American National Standards Institute, August 1994.

[16] C. Mohan, H. Pirahesh, W. Tang, and Y. Wang. Parallelism in relational database management systems. In *IBM SYSTEMS JOURNAL, Vol 33, No 2*, pages 349–371, 1994.

[17] Tam Nguyen and V. Srinivasan. Accessing relational databases from the world wide web. In *SIGMOD 96*, pages 529–540, 1996.

[18] Oracle. *Oracle 7 Server Application Developer's Guide*, Dec. 1992. Part: 6695-70-0212.

[19] UniSQL. *UniSQL Object-Relational Database Technology*, 1996. White paper by Dr. Won Kim.