Paul Watson

Cell: +1 913 387 7529
Email: paul@oninit.com

Agenda

Simple UDRs

Registering and software ‘theft’

Internal Magic

SQL and Cursors

Iterators

Tracing and Debug

The Myths

Black Art

It's just C

Difficult

It's just C

they can be a simple one line statement

they can be a close to an entire application

Crash the engine

It's just C

Like any other C program, a bad program is a bad program. Unfortunately, a bad C UDR will definitely, 100% crash the engine

Simple UDRs

```
void udr_srand(mi_integer seed)
{
        srand(seed);
}

mi_integer udr_rand()
{
        return rand();
}


mi_integer udr_get_randmax()
{
        return RAND_MAX;
}
```

C UDR are (always) faster

```
mi_boolean bit_pos_on(mi_integer item, mi_integer bitpos)
{
   long mask = 1 << (bitpos-1);
   return ((mask & item) == mask) ? '\1' : '\0';
}
```

Check C function speed
select count(*) from checks
where bit_pos_on(tablemaskid, 15)
Time: 7.876 seconds

Check speed of SYSMASTER:bitval
select count(*) from checks
where bitval(tablemaskid, 16384) = 1
Time: 12.578 seconds

Pet Hate:
DO NOT USE BOOLEAN DATATYPES
the datatype of Satan

Registering a UDR

```
begin work;

create function if not exists
jaro_winkler_distance(lvarchar, lvarchar) returning FLOAT
      with (handlesnulls, class="oninit")
      external name
"$INFORMIXDIR/extend/Oninit.1.0/oninit.bld(jaro_winkler_dist
ance)"
       language C
end function;

grant execute on function jaro_winkler_distance(lvarchar) to
public;

commit work;
```

https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance

jaro_winkler_distance

The Jaro–Winkler distance is a string metric measuring an edit distance between two sequences

Built into Oracle

```
double jaro_winkler_distance(const char *s, const char *a)
{
    int i, j, l;
    int m = 0, t = 0;
    int sl = strlen(s);
    int al = strlen(a);
    int sflags[sl], aflags[al];
    int range = max(0, max(sl, al) / 2 - 1);
    double dw;
……….
```

jaro_winkler_distance

Now you could re-write the code to be directly accessible to the engine but, generally, the simplest way is to write a wrapper

```
mi_double_precision *jarod_winkler_distance(mi_lvarchar *lv1,
mi_lvarchar *lv2, MI_FPARAM *fparam_ptr)
…
if ((mi_fp_argisnull(fparam_ptr, 0)) ||
(mi_fp_argisnull(fparam_ptr, 1)))
        mi_db_error_raise(NULL, MI_EXCEPTION, "NULL parameter
disallowed");

s1 = mi_get_vardata(lv1); l1 = mi_get_varlen(lv1); s1=mi_lvarchar_to_string(lv);l1=strlen(s1);
s2 = mi_get_vardata(lv2); l2 = mi_get_varlen(lv1); s1=mi_lvarchar_to_string(lv);l2=strlen(s2);

retdbl=jaro_winkler_distance(s1, l1, s2, l2);
retptr = mi_dalloc(sizeof(mi_double_precision), PER_ROUTINE);
retptr  = retdbl;
return retptr;
```

Random Numbers - Makefile

```
default : udr_rand.bld
include $(INFORMIXDIR)/incl/dbdk/makeinc.linux86_64


CC = gcc
CFLAGS = -W -Wall -ansi -pedantic -O2
CBLDINCLS = -I${INFORMIXDIR}/incl/public -I${INFORMIXDIR}/incl/esql -
I${INFORMIXDIR}/incl
CBLDFLAGS = -DMI_SERVBUILD -fPIC -shared
LDBLDFLAGS = -shared -u _etext
.SUFFIXES:
.SUFFIXES: .o .c

.c.o:
        ${CC} ${CFLAGS} ${CBLDFLAGS} ${CBLDINCLS} -c $< FILES.c = udr_rand.c
FILES.o = ${FILES.c:.c=.o}

udr_rand.bld:
        ${FILES.o} ${LD} ${LDBLDFLAGS} -o $@ ${FILES.o}

clean:
        ${RM} ${FILES.o} udr_rand.bld
```

https://www.oninit.com/reference/index.php?id=random.html

Random Numbers - Register

```
create function udr_get_randmax()
        returns integer
        with (class="rnd") external name
"$INFORMIXDIR/extend/rnd/udr_rand.bld(udr_get_randmax)"
language c;



VPCLASS
        VPCLASS=rnd
```

https://www.oninit.com/reference/index.php?id=random.html

Makefile Gotchas

Be careful if you are using Informix provided datablade functions as sometimes these are tightly coupled to the engine.

For example Timeseries

Add to CFLAGS

```
-I$(INFORMIXDIR)/extend/TimeSeries.6.00.$(VERSION)/lib
```

Set LIBS

```
$(INFORMIXDIR)/extend/TimeSeries.6.00.$(VERSION)/lib/tsbeapi.a
```

Remember to add MI_SERVBUILD to the gcc

MI_FPARAM

When the database server calls a user-defined routine, it passes an extra argument, MI_FPARAM, to the routine. The MI_FPARAM structure contains state information about the routine. Typically, you use the MI_FPARAM argument to test whether another argument is null, to set another argument to NULL, or to store iterated values. You access the values contained in the MI_FPARAM structure by using the accessor functions provided by the DataBlade API.

The structure can tell us how parameters where passed, where they are null or not, maintain a user defined state etc

MI_FPARAM

```
mi_integer udr_get_randmax(MI_FPARAM *fparam)
{
        (void) fparam;

        return RAND_MAX;
}

create function udr_get_randmax()
returns integer
with (class="rnd")
external name
"$INFORMIXDIR/extend/rnd/udr_rand.bld(udr_get_randmax)"
language c;
```

mi_fparam_get_current() can be used to get the MI_FPARAM structure

MI_FPARAM

mi_fp_argisnull()

The accessor function determines whether the argument of a user-defined routine is an SQL NULL value from its associated MI_FPARAM structure.

mi_fp_funcstate()

The accessor function obtains user-state information for the user-defined routine from its associated MI_FPARAM structure.

Memory

Always use the published APIs, never use the standard C calls.

All the mi_<memory> functions are macros to the same underpinning function

Memory duration

```
PER_ROUTINE      PER_COMMAND   PER_STMT_EXEC PER_STMT_PREP
PER_TRANSACTION PER_SESSION   PER_SYSTEM
```

Global and Module Variables

Do Not use Global or Module variables

```
char  myval[100] = "my initial string";

void my_func1()
        strcpy(myval, "my second string");
        return myval;

void my_func2()
        return myval;

execute procedure my_func1();
execute procedure my_func2();  << could crash the engine
execute procedure my_func1();
```

Named Memory

Named memory is memory allocated from the database server shared memory, as user memory. You can, however, assign a name to a block of named memory and then access this memory block by name and memory duration. The database server stores the name and its corresponding address internally. By contrast, user memory is always accessed by its address.

This functionality allows 'global' variables based on the memory duration.

The engine will not stop multiple threads writing to the same named memory area, if in scope, that is the responsibility of the UDR

Note the unque'ness is name AND duration so the name can be duplicated

```
ptr1 = mi_named_get(PER_SYSTEM,  "my_memory")
ptr2 = mi_named_get(PER_SESSION, "my_memory")
ptr1 != ptr2 /* and never will */
```

Fastpath

How to access/execute a UDR from within a UDR.

For example,

```
create procedure tscontainercreate (lvarchar,lvarchar,lvarchar,int,int)
external name
"$INFORMIXDIR/extend/TimeSeries.6.00.FC4/TimeSeries.bld(ts_make_container)"
language c;
```

Slow way

Use mi_exec and the SQL interface

Fast way

Fastpath allows you register a function so it can be called directly.

```
func_desc = mi_routine_get(conn, 0, "tscontainercreate(mi_lvarchar, ..)");
mi_routine_exec(conn, func_desc,   &error, arg1, ……);
```

The Fastpath interface is a facility that allows the execution of UDR functions without having to go through the parser, the optimizer and the executor

Questions

SQL and Cursors

If only we had ‘FOREACH’ logic ….

SQL and Cursors

We need to write the heavy lifting that is hidden when using 4gl, ESQL, SPL or any other drivers

However, with simple statements, there is not a lot to lift.

```
sprintf(sqlstr,"SELECT * FROM %s WHERE 1=0 INTO TEMP mytab WITH NO LOG", MYTAB);
if(MI_ERROR==(mi_exec(CONN, sqlstr, MI_QUERY_NORMAL)))
       MI_DB_ERROR_RAISE(NULL, MI_EXCEPTION, "oni_clobtotable: failed to create
internal table");

sprintf(sql_ts,"execute function
tsl_init('%s','%s',1,2,'/tmp/tsl.log')",MYTAB,MYTS);
if(MI_ERROR==(mi_exec(CONN,sql_ts,MI_QUERY_NORMAL)))
     MI_DB_ERROR_RAISE(NULL, MI_EXCEPTION, "oni_clobtotable: TSL Attach
failed");
```

SQL and Cursors

Prepared statements, again not a lot of heavy lifting

```
mysql=oni_getsql(udrcb, MYLF, sqllf, MYLTST, MYPK);
if(NULL==(stmt_lf=mi_prepare(CONN,sqllf,NULL)))
     MI_DB_ERROR_RAISE(NULL, MI_EXCEPTION, "oni_clobtotable: failed to prepare
stmt_lf[2]");
```

But what is the equivalent of

```
EXEC SQL execute upd_id using :l_var1, :l_var2;
```

SQL and Cursors

```
mi_integer mi_exec_prepared_statement(stmt_desc, control,
   params_are_binary, n_params, values, lengths, nulls,
types, retlen,
   rettypes)
   MI_STATEMENT *stmt_desc;
   mi_integer control;
   mi_integer params_are_binary;
   mi_integer n_params;
   MI_DATUM values[];
   mi_integer lengths[];
   mi_integer nulls[];
   mi_string *types[];
   mi_integer retlen;
   mi_string *rettypes[];
```

Statement prepared elsewhere

Return data as binary format

What is format of the passed variables

How many parameters are being passed

Values of the parameters

Lengths of parameters

Are there NULL parameters

What the datatypes

How many values are returns

What are the returned datatypes

SQL and Cursors

```
mi_integer mi_exec_prepared_statement(stmt_desc, control,
   params_are_binary, n_params, values, lengths, nulls,
types, retlen,
   rettypes)
   MI_STATEMENT *stmt_desc;
   mi_integer control;
   mi_integer params_are_binary;
   mi_integer n_params;
   MI_DATUM values[];
   mi_integer lengths[];
   mi_integer nulls[];
   mi_string *types[];
   mi_integer retlen;
   mi_string *rettypes[];
```

values = “mystr”, NULL, “22”
lengths = 5,0,2
nulls = 0,1,0
types = “lvarchar”,”date”,”integer”

Aside: MI_DATUM

The DataBlade API handles a generic data value as an **MI_DATUM** value, also called a *datum*. A datum is stored in a chunk of memory that can fit into a computer register.

In the C language, the **void *** type is a typeless way to point to any object and will hold any integer value. This type is usually equivalent to the **long int** type and is usually four bytes in length, depending on the computer architecture. **MI_DATUM** is defined as a **void *** type.

The **MI_DATUM** data type is guaranteed to be the size of the C type **void *** on all computer architectures.

On 64-bit platforms, **void *** is eight bytes in length, so an **MI_DATUM** value is stored in eight bytes.

SQL and Cursors

But what is the equivalent of

```
OPEN c_cc;
        FETCH c_cc INTO cnt;
CLOSE c_cc;
```

This time we need to use mi_open_prepared_statement()

Very similar to mi_exec_prepared statement but know we can define the cursor type in the control flag [read-only sequential cursor, update scroll cursor or read-only scroll cursor]

We can also name the cursor

But there is a lot heavy lifting

SQL and Cursors

```
mi_open_prepare_statement   != MI_ERROR
mi_fetch_statement != MI_OK
if(MI_ROWS != mi_get_result())
        No rows
else
        while (MI_NO_MORE_ROWS != (result=mi_get_result())
        case
                MI_ERROR:    ……
                MI_DDL: ……..                          If SELECT/procedure or function then MI_DDL and
                MI_DML: …….                           MI_DML are invalidate flags, trap appropriately
                MI_ROWS:
                        for (i = 0, i< numcols; i++)
                                mi_value(row, &datum, &len)
                                if(datum!=NULL)
                                        savedata to …..
                                else
                                        process null condition
                                endif
                default: assert(the engine is broke)
        end cae
endif
```

Iterators

By default UDRs return one value, or single row. An iterator function allows the UDR to return multiple values. Think RESUME within SPL.

An iterator has 3 distinct phases and they are detected via the mi_fp_request function and is typically processed as a case statement

```
state = mi_fp_request(p_fparm)
        case state
                SET_INIT
                SET_RETONE
                SET_END
```

Iterators – SET_INIT

The initial phase, is used to configure what will be needed throughout the running of the UDR, such as memory, cursors etc.

```
mydata  = mi_alloc(sizeof(my_complex_struct));
cfg_lots_stuff(mydata);
mi_fp_setfuncstate(p_fparam, (void *) mydata);
```

This tells the engine that the iterator is ‘good to go’

Note: this doesn’t return anything

Iterators – SET_RETDONE

The second phase, does all the 'heavy lifting'

```
mydata = mi_fp_funcstate(p_fparam);
If something
        process mydata
        mi_fp_setfuncstate(f_fparam, (void *)mydata)
        return value(s)
else
        mi_fp_setisdone(p_fpraram, MI_TRUE)
fi
```

Iterators – SET_END

The final phase is to tidy up what was setup under SET_INIT

```
mydata = mi_fp_funcstate(p_fparam);
mi_free(mydata)
mydata =NULL
```

Note: this doesn't return anything

Questions

Replacing a UDR

This is not straightforward as copying the files to the correct location. In fact, just overwriting the existing library will probably crash the engine.

Easiest Method

Bounce the engine

Difficult way

```
execute function
ifx_unload_module("$INFORMIXDIR/extend/Oninit.1.0.FC3/oninit
.bld(oni_clobtotable)",'C');
```

Tracing

Unlike C programs there is no output to screen, so we need a mechanism to capture what is going on.

Systraceclass

```
macro  DPRINTF
```

```
DPRINTF("oni_statistics", 1, ("%s: the main function"I, __func__));
DPRINTF("oni_table_stats_sql", 11, ("%s: Stats SQL %s", __func__, sqlstr));
DPRINTF("oni_table_stats_chk", 21, ("%s: Last Cnt: %d Curr Cnt :%d Thres %4.2f),
__func__, last_cnt, curr, threshold);
```

Turning on Tracing

```
void oni_trace(mi_lvarchar *trace_path, mi_lvarchar *trace_level)
{
          mi_tracefile_set(mi_lvarchar_to_string(trace_path));
          mi_tracelevel_set(mi_lvarchar_to_string(trace_level));
}

begin work;
    delete from  systraceclasses where name = "oninit";

    insert into systraceclasses values("oninit",11);

    drop procedure if exists oni_trace(LVARCHAR, LVARCHAR);

    create procedure oni_trace(LVARCHAR, LVARCHAR)
        external name "$INFORMIXDIR/extend/Oninit.1.0/oninit.bld(oni_trace)"
LANGUAGE C;

    grant execute on procedure oni_trace (lvarchar, lvarchar) to public;

commit work;
```

Conditional Tracing

To perform computations for a tracepoint, define a trace block.

A trace block initially compares a specified threshold with the current trace level to determine whether to continue with the trace computations. Within the trace block, it can output a result in a trace message.

```
if( tf("oni_trace", 30) )
{
    PurgeFactor= oni_purge_det(oni_udrcd->ts->tsid,  oni_udccb->ts->clob_data);

   tprintf("%s: x = %d and consFactor = %f", __func__, oni_udrcb->ts->tsid, Purgefactor);
}
```

Note: gl_tprintf will provide GLS functionality

Debugging

Very simple UDR can be converted into a standalone program

The af stack trace is your friend

Scatter DEBUG code throughout the UDR that is activated when compiled

```
#define DEBUG

#ifdef DEBUG
          lots of debug data
#endif
```

Add DPRINTF call before you exit the UDR

Debugging

As most UDRs cannot be converted into a standalone program 'Real' UDRs take more effort to debug.

- VPCLASS=myclass,num=1 [maybe noyield as well]
- Compile with -ggdb3 option to use gdb
- onstat –g glo to find the PID of the VP
- gdb –p <PID>

99.9% of errors will be your fault, memory issues, buffer overruns, NULLs etc.

Aside: just running running gdb on the oninit start up can be enlightening

Useful things to include

Make sure you UDR has version information

```
mi_integer oni_version() {    return (UDRVERSION);}
```

Before the exit point in the UDR always have a DPRINTF

Prefix any external function with something unique to you/your company etc

If you can then operate with Informix binary data

Questions