14. Coding User-Defined Functions

O DBMaker permite que programadores criem suas próprias funções definidas pelo usuário (UDFs). Uma vez que uma UDF é escrita em DBMaker, ela é tratada como uma nova função interna do DBMaker com os mesmos usos. Criar uma nova função definida pelo usuário é simples e segue o procedimento geral descrito abaixo.

Para criar uma função definida pelo usuário

1. Escreva uma função definida pelo usuário em C (interface UDF)
a) a) Escreva a instrução include
b) b) Escreva o cabeçalho da função
c) c) Escreva os argumentos que a função passa
d) d) Defina memória alocada, se necessário
e) e) Defina um código de erro, se desejado
2. Compile a biblioteca de vínculo dinâmico (DLL) para a UDF
3. Crie a UDF no DBMaker, informando o array de dados a ser passado para a UDF

14.1 UDF Interface

O primeiro passo para criar uma UDF é codificá-la em C. As seções a seguir fornecem um exemplo de uma UDF em C e descrevem cada um dos elementos do código específicos de uma UDF do DBMaker.

Exemplo:

Se você deseja criar uma nova UDF, INT2STR(), para converter dados inteiros em string, deverá construir uma biblioteca de vínculo dinâmico (DLL) para incluí-la.

```
dmSQL> SELECT INT2STR(c1) FROM t1; // c1 is integer type
```

O código fonte C a seguir, template.c, fornece um exemplo do código da UDF INT2STR().

```
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include "libudf.h"
/* Transfer integer type to string type */
```

```
#ifdef WIN32
__declspec(dllexport)
#endif
int INT2STR(int narg, VAL args[])
{
 char *ptag;
 int len;
 char p1[11];
 int rc;
 if (args[0].type != NULL_TYP)
 {
 sprintf(p1, "%d", args[0].u.ival);
len = strlen(p1);
 if (rc = _UDFAllocMem(args, &ptag, len))
 return rc;
 memcpy(ptag, p1, len);
 args[0].type = CHAR_TYP;
 args[0].len = len;
 args[0].u.xval = ptag;
 }
 return _RetVal(args, args[0]);
}
```

Including libudf.h

O DBMaker define algumas constantes, tipos de dados e interfaces comuns, necessários para a codificação de UDFs.

Programadores devem incluir o arquivo libudf.h antes de qualquer codificação de UDF.

```
#include "libudf.h"
```

Passing Parameters

Os argumentos de uma UDF usada em um comando SQL são agrupados no parâmetro args da UDF codificada na linguagem C. Através do array args, uma UDF obtém os dados de entrada. args também é chamado de bloco de controle da UDF, sempre usado como o primeiro argumento da interface comum fornecida pelo DBMaker. Algumas interfaces comuns, como a Interface Comum BLOB, serão introduzidas posteriormente.

Cada cabeçalho de UDF em uma função C deve seguir o formato:

```
int FUNCTION_NAME(int narg, VAL args[])
{
...
}
```

NOTA: args[] aponta para um vetor. Funções que recebem apenas um argumento devem usar a forma com ponteiro: *args.narg especifica quantos argumentos a função passa. Por exemplo, se uma UDF (Função Definida pelo Usuário) MYSUBSTRING (c1, c2, c3) é chamada em um comando SQL, a informação de c1 é passada por args[0], c2 por args[1] e c3 por args[2]. O valor de narg, que especifica o tamanho do vetor, é 3.

- Exemplo 1:

Considerando o valor de c1 como 'abcdefghijklmn', args[0] seria:

```
args[0].type = CHAR_TYP
args[0].len = 14
args[0].u.xval = "abcdefghijklmn"
```

- Exemplo 2:

Se o valor de c2 for o inteiro 30, args[1] seria: **30**

```
args[1].type = INT_TYP
args[1].len = 4
args[1].u.ival = 30
```

Além de CHAR_TYP e INT_TYP, as constantes BIN_TYP, FLT_TYP, OID_TYP, BLOB_TYP, DEC_TYP e NULL_TYP são definidas em libudf.h.

```
#define BIN_TYP 0x0000 /* bit string data type*/
#define CHAR_TYP 0x1000 /* character data type*/
#define INT_TYP 0x2000 /* integer data type*/
#define FLT_TYP 0x3000 /* floating point data type*/
#define OID_TYP 0x4000 /* OID data type*/
#define BLOB_TYP 0x5000 /* BLOB data type*/
#define DEC_TYP 0x6000 /* decimal data type*/
#define NULL_TYP 0xF000 /* set if column is null */
```

- Exemplo 3:

Através de NULL_TYP, o programador pode saber se o dado de entrada é NULL (vazio).

```
if (args[0].type == NULL_TYP)
{
 /* input data is NULL */
}
else
{
 /* input data is not NULL */
}
```

A estrutura de dados completa de VAL, conforme definido em libudf.h:

```
typedef struct tVAL {
 i16 type; /* data type */
 i15 len; /* data length */
 union {
 i31 ival; /* long integer data */
 i15 sival; /* short integer data */
 double fval; /* double data */
 float sfval; /* float data */
 dec_t dval; /* decimal data */
 char *xval; /* pointer to data */
 } u;
} VAL;
```

A estrutura dec_t, usada para o tipo DECIMAL, em libudf.h:

```
typedef struct
{
 i8 pre;
 i8 sca;
 i8 dgt[20];
i8 exp;
i8 junk;
} dec_t10;
typedef dec_t10 dec_t;
```

Uma UDF (Função Definida pelo Usuário) não apenas passa dados de entrada através do tipo VAL, mas também retorna dados de saída por ele. A maneira de retornar dados será discutida posteriormente.

Allocating Memory Space

Em funções C, você pode precisar alocar memória e liberá-la antes de sair da função. Valores retornados, como strings ou IDs temporários de BLOB, precisam alocar memória, mantê-la na função UDF e ter o DBMaker auxiliando na liberação do espaço de memória.

EXEMPLO:

No exemplo a seguir de uma UDF (Função Definida pelo Usuário) UDFAllocMem, arg é o bloco de controle da UDF, ppt é o ponteiro para obter o bloco de memória alocada e nb é o tamanho alocado desejado. Esta função aloca memória e a mantém até o DBMaker liberar a memória. O DBMaker se encarrega disso.

```
int _UDFAllocMem(VAL *arg, char **ppt, int nb);
```

O DBMaker sabe liberar a memória após o retorno de um resultado usando args[0].u.xval, que é um ponteiro para o espaço de memória alocado pela função _UDFAllocMem().

```
if (rc = _UDFAllocMem(args, &ptag, 10))
 return rc; /* return error code */
memcpy(ptag, "0123456789", 10);
args[0].type = CHAR_TYP;
args[0].len = len;
args[0].u.xval = ptag;
```

Returning Results

Há dois tipos de valores retornados: um é um código de erro e o outro é o resultado da UDF através do tipo de argumento VAL. Códigos de erro são retornados ao DBMaker, mas seus valores são ocultados do usuário; apenas uma mensagem de erro será exibida. A seguir, descrevemos como os códigos de erro são retornados.

O cabeçalho da UDF em uma função C segue a forma:

```
int FUNCTION_NAME(int narg, VAL args[]);
```

Se FUNCTION_NAME() retornar um valor diferente de zero, significa que há algum problema. Se 0 for retornado, indica que a função funcionou corretamente.

Antes de sair da UDF, chame _RetVal() para passar o resultado obtido pela UDF para o DBMaker. A declaração da função _RetVal() é a seguinte:

```
int _RetVal(VAL *arg, VAL rtn);
```

O primeiro argumento arg é o bloco de controle da UDF (função definida pelo usuário) e o segundo argumento rtn é o valor retornado. O código a seguir retorna o número inteiro 30:

```
int rc; /* error code */
VAL rtn;
rtn.type = INT_TYP;
rtn.len = 4;
rtn.u.ival = 30;
rc = _RetVal(arg, rtn); /* pass result back to DBMaker */
return rc; /* return error code (0 means no error) */
```

14.2 Building UDF Dynamic-Link Library

O DBMaker fornece uma biblioteca dmudf.lib para vincular com o arquivo fonte da UDF e criar a biblioteca de ligação dinâmica. Como a biblioteca de ligação dinâmica é diferente nos ambientes Microsoft Windows e UNIX, ambos os casos serão discutidos separadamente.

DLL in Microsoft Windows Environment

O DBMaker também fornece o código fonte template.c no diretório /udf_templates e os arquivos modelo de compilação udf42.mak (para Microsoft VC++ versão 4.2), udf50.mak (para Microsoft VC++ versão 5.0) ou udf60.mak (para Microsoft VC++ versão 6.0) para usuários WIN32. Para usuários WIN32 e WIN64 do Microsoft Visual Studio 2005 e superior, há o arquivo modelo udf80.mak. Você pode seguir o formato de um arquivo fonte C de modelo para escrever sua UDF.

Nos exemplos/instruções a seguir, o arquivo udf60.mak é utilizado.

1. Certifique-se de onde incluir o arquivo dmudf.lib e, em seguida, use o IDE fornecido pelo Visual C++ para modificar as alterações necessárias.

2. Copie o arquivo de modelo udf60.mak para o diretório desejado e renomeie-o com um nome de arquivo de compilação (makefile).
3. Escolha a opção no menu para abrir o workspace do projeto do arquivo makefile. (substitua "a opção" pelo comando específico do menu, por exemplo, "Abrir Projeto")
4. Escolha a opção no menu e clique em template.c. Para remover o template, clique em "Excluir". (substitua "a opção" pelo comando específico do menu)
5. Escolha o item na barra de ferramentas, selecione "Adicionar ao Projeto", selecione "Arquivos Existentes" e, em seguida, insira seu próprio arquivo .c no arquivo makefile do workspace do projeto.
6. Em Configurações do Projeto -> Configurações de Gerenciamento de Configurações, escolha "WIN32 Debug" para este exemplo. Nas Configurações do Projeto, você pode alterar os diretórios de saída. No arquivo makefile template, defina 60Deb como os diretórios intermediário e de saída.
7. Nas Configurações do Projeto, na categoria "Linker", altere o nome do arquivo .dll diretamente na opção "Saída". Além disso, altere o caminho de link do arquivo dmudf.lib que o DBMaker suporta na opção "Entradas" para o diretório de trabalho.

Após concluir as etapas acima, você pode criar seu próprio arquivo makefile para DLL. Seguindo etapas semelhantes, também é possível criar um arquivo makefile para uma DLL da versão WIN32 Release.

Usuários do VC++ também podem criar um arquivo makefile para DLL usando as mesmas etapas, mas definindo o alinhamento dos membros da estrutura como 4 bytes. No workspace do projeto IDE VC++ 6.0, selecione o item de menu C/C++ e, em seguida, na caixa de diálogo Categoria, escolha **Propriedades de Configuração**. Você encontrará a opção de alinhamento de membros da estrutura e poderá escolher 4 bytes como resultado.

Ao escrever uma DLL usando o arquivo makefile como base, observe as configurações nele definidas. Se você não quiser usar template.c como o nome padrão do arquivo C dentro do makefile, remova template.c do udf60.mak e insira seu próprio arquivo C no workspace do projeto udf60.mak.

Exemplo:

No código DBMaker template.c, lembre-se de incluir o arquivo libudf.h fornecido e exportar suas funções. Use o método de função de exportação do guia do programador VC++ ou o seguinte:

```
__declspec(dllexport) datatype YOUR_FUNCTION_NAME( ..... )
```

Como alternativa, você pode criar um arquivo def no workspace do projeto para exportar suas funções. Lembre-se que o nome da função para a UDF (função definida pelo usuário) deve estar em LETRA MAIÚSCULA no código fonte C.

Após concluir as etapas acima, você pode compilar um arquivo dll de versão debug/release, criando assim um arquivo udf60.dll.

Nos instruções seguintes, udf80.mak é usado.

1. Copie udf80.mak e udf80.def, template.c e udf60.dsp para o diretório desejado e renomeie udf60.dsp para udf80.dsp.
2. Edite udf80.def. Você pode substituir template.c pelo seu próprio arquivo .C.
3. Edite udf80.def. Você pode alterar os diretórios de saída. No arquivo make do modelo, defina 60Deb como os diretórios intermediário e de saída.
4. Edite udf80.def. Você pode alterar o nome do arquivo .dll de saída diretamente para udf80.mak. Depois de concluir as etapas acima, você pode criar seu próprio arquivo make dll com cmd.

Abra o prompt de comando, navegue até o diretório desejado usando o comando cd e então execute o seguinte comando.

```
@CALL bat_vs_env
@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32
Debug"> make.ms
```

bat_vs_env: Seu valor é determinado pela versão do Visual Studio e pelo sistema operacional. Por exemplo, se a compilação C for do VS2005 e o sistema operacional for de 32 bits, substitua bat_vs_env por "C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat". Para obter detalhes, consulte a tabela a seguir:

VS Vision	OS	bat_vs_env
VS2005	32	C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat
	64	C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\amd64\vcvarsamd64.bat

VS2008	32	C:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat
	64	C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat
VS2010	32	C:\Program Files\Microsoft Visual Studio 10.0\VC\bin\vcvars32.bat
	64	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat
VS2012	32	C:\Program Files\Microsoft Visual Studio 11.0\VC\bin\vcvars32.bat
	64	C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\amd64\vcvars64.bat

Para construir a versão Release da dll, você pode substituir Debug por Release na linha de comando "@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG=""udf80 - Win32 Release"" > make.msg".

UDF so File in UNIX

Uma biblioteca dinâmica UNIX, também conhecida como arquivo ".so", pode ser criada.

Exemplo:

Escreva o código fonte C UDF, no exemplo o arquivo é chamado udf.c. Após a conclusão, use a função UDF em um sistema operacional baseado em UNIX.

```
$ cc -c udf.c
$ ld -o libudf.so udf.o -lm
$ dmsqlt
dmSQL> CREATE FUNCTION libudf.INT2STR(INT) RETURNS CHAR(10);
```

NOTA: As opções do comando `ld` no exemplo acima podem variar dependendo do sistema UNIX. Elas podem ser `-G`, `shared` ou algo diferente. Consulte o manual do seu sistema UNIX ou as páginas `man` para verificar como usar o comando `ld` na construção de uma biblioteca compartilhada.

14.3 Creating, Using, and Dropping UDF

A próxima etapa para uma função definida pelo usuário (UDF) é criá-la dentro do DBMaker. As seções a seguir descrevem a sintaxe para criar, consultar e excluir uma UDF.

Creating a UDF Sintaxe:

```
dmSQL> CREATE FUNCTION <udf_dll_name.function_name>
(<function_datatype>) RETURNS
<function_output_datatype>;
```

Querying a UDF

Sintaxe:

```
dmSQL> SELECT <function_name>(<related_table_column_name>)
FROM <related_table>;
```

Dropping a UDF Sintaxe

```
dmSQL> DROP FUNCTION <function_name>;
```

Exemplo:

O seguinte demonstra como usar um arquivo UDF.

Exemplo 1:

Considerando um banco de dados chamado DMDEMO contendo uma tabela, tb_UDF, com a estrutura da tabela: número inteiro (INT) e nome caractere (CHAR(10)).

```
dmSQL> SELECT * FROM tb_UDF;
number name
======== =========
10          1
20          2
30          3
3 rows selected
```

Usando o modelo de exemplo template.c fornecido pelo DBMaker, agora podemos construir um udf60.dll com sucesso.

No arquivo dmconfig.ini, adicione uma linha na seção DMDEMO:

```
[DMDEMO]
DB_DbDir = D:\\UDFDEMO
DB_FoDir = D:\\UDFDEMO\\FO
DB_LbDir = D:\\UDF\\60Deb ; add this line
```

Para obter mais informações sobre DB_LbDir, consulte Palavras-chave em dmconfig.ini. Defina DB_LbDir ou coloque o udf60.dll em \shared\udf, pois é o diretório padrão do UDF.

Exemplo 2:

Inicie o banco de dados DMDEMO e crie a função UDF. No exemplo, o arquivo DLL é udf60, a função é INT2STR, o parâmetro de entrada é INT e o parâmetro de saída é CHAR(10).

```
dmSQL> CREATE FUNCTION udf60.INT2STR(INT) RETURNS CHAR(10);
```

A função UDF **INT2STR** retorna os seguintes resultados. O **<function_name>** é **INT2STR**, o **<related_table_column_name>** é **number** de acordo com o esquema de **tb_UDF** e o **<related_table>** é **tb_UDF**.

```
dmSQL> SELECT INT2STR(number) FROM tb_UDF;
INT2STR(number)
==================
10
20
30
3 rows selected
```

Exemplo 3: Outra função UDF, por exemplo, **STR2INT()**, no mesmo arquivo de link dinâmico:

```
Another UDF function, e.g., STR2INT(), in the same dynamic-link
file:
dmSQL> CREATE FUNCTION udf60.STR2INT(CHAR(10)) RETURNS INT;
dmSQL> SELECT STR2INT(name) FROM tb_UDF;
```

```
STR2INT(name)
==============
1
2
3
3 rows selected
```

Exemplo 4: Ao excluir uma função UDF, basta excluir o nome da função UDF; não é necessário anexar o nome do arquivo DLL da UDF. Ao excluir uma função UDF, aguarde até que o banco de dados seja encerrado, e então a função UDF será removida. Antes que o banco de dados seja encerrado, a função continuará a existir.

```
dmSQL> DROP FUNCTION INT2STR;
```

14.4 Create XML Validate UDF

Flexml

O **flexml** de Kristoffer Rose, distribuído sob a Licença Pública Geral GNU, é um gerador de processos XML. Ele utiliza um arquivo DTD e gera um arquivo LEX. O **flexml** está disponível em http://flexml.sourceforge.net.

GENERATING THE LEX FILE

```
$                              flexml                                  name.dtd
```

O fluxo de entrada original do LEX é um fluxo de entrada de **FILE**. O arquivo LEX deve ser modificado para usar **UDF Blob** como fonte de entrada. O exemplo a seguir demonstra essa modificação adicionando um **YY_INPUT** personalizado. Exemplo: Modifique a seção de definição do arquivo LEX adicionando **YY_INPUT** conforme mostrado abaixo. A seção de definição está localizada no início do arquivo, entre os marcadores **"%{"** e **"}%"**.

```
#include "libudf.h"
typedef struct udf_file
{
VAL *args;
i31 handle;
```

```
i31 rc;
 i31 left;
i31 rlen;
} UDF_FILE;
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) {\\
 UDF_FILE * uf =(UDF_FILE *)yyin; \\
errno=0; \\
if ( uf->left <= 0 )\\
{\\
result = (uf->rlen=0);\\
}\\
if ( (uf->rc = _UDFBbRead(uf->args, uf->handle, max_size, &(uf-
>rlen),
buf))!=0 ) \\
{ \\
 errno = uf->rc; \\
 result = 0;\\
}\\
else\\
{\\
 uf->left -= uf->rlen;\\
 result = uf->rlen;\\
}\\
}\\
```

Em seguida, adicione a função UDF ao final do arquivo LEX conforme mostrado abaixo:

```
#ifdef WIN32
__declspec(dllexport)
#endif
int XXX_VALIDATE(int nArg, VAL args[])
{
 BBObj bbin;
 UDF_FILE uf;
 int rc = 0;
 int rc2 = 0;
 memset(&uf, 0, sizeof(UDF_FILE));
 memcpy((char *)(&bbin), args[0].u.xval, BBOBJ_SIZE);
 uf.args = args;
```

```
 rc = _UDFBbOpen( args, bbin, &(uf.handle));
if( rc != 0 )
 goto EXIT;
 if (rc = _UDFBbSize(args, bbin, &(uf.left)) )
{
 goto EXIT;
}
 yyin = (void *)&uf;
 rc = validdtd00udf();
 rc2 = _UDFBbClose(args, uf.handle);
EXIT:
 if( args[0].type != NULL_TYP ) // null column data
{
 args[0].type = INT_TYP;
 args[0].len = 4;
 args[0].u.ival = (rc == 0? 1:0);
}
 return _RetVal(args, args[0]);
}
```

BUILD DLL/SO

```
lex                                                                    name.l
cc   -c   -DBUILD_DLL   lex.name.c   -Idbmaker-installed-dir/include
```

CREATE UDF

```
dmSQL>   CREATE   FUNCTION   dllname.udfname(BLOB)   returns   int;
```

CREATE COLUMN WITH CHECK CONSTRAINT

```
dmSQL> CREATE TABLE table-name( c1 XMLTYPE CHECK udfname(value) = 1);
```

DBMaker DTD Validation UDF Generator

Ferramenta de linha de comando para gerar uma UDF de validação para o DTD especificado. O nome do arquivo DTD é obrigatório; se não for especificado, um erro será gerado. O diretório de saída é opcional; se não for especificado, os arquivos serão

criados no diretório de trabalho atual. O prefixo é opcional; se especificado, o arquivo gerado usará o prefixo no nome do arquivo. Se não for especificado, o nome do arquivo DTD sem a extensão será utilizado.

```
$ dmxmludfmk –dtd dtd-file-name [–o output-directory] [–p prefix]
```

Vários arquivos são gerados conforme segue:

- Arquivo Lex: **<prefixo-especificado-pelo-usuário.l>**
- Arquivo Yacc: **<prefixo-especificado-pelo-usuário.y>**
- Arquivo de função UDF: **<prefixo-especificado-pelo-usuário> udf.c** e **<prefixo-especificado-pelo-usuário> udf.h**
- A função UDF é nomeada como **<prefixo-especificado-pelo-usuário>_VALIDATE**
- **hash.c** e **hash.h** fornecem funções de hash
- **Makefile** para plataformas UNIX ou **Makefile.msvc** para plataformas Windows

Exemplo 1:

```
Make <user-specified-prefix>.so ;for UNIX
```

Exemplo 2:

```
Nmake /f Makefile.msvc ;for Windows visual studio 2005 or 2008
```

Exemplo 3:

```
nmake /f Makefile.msvc COMPILER=VC60 ;for Windows visual studio 6.0
```

Exemplo 4:

```
nmake /f Makefile.msvc OSTYPE=$OSTYPE ; for cygwin environment
```

Observe que **dmxmludfmk** suporta **ASCII**, **BIG5**, **gb**, **shiftJIS**, e **utf8**, enquanto **flexml** suporta substituição de conteúdo apenas para entidades DTD definidas internamente

Default Validator

I_VALIDATE é fornecido como um validador padrão para verificar a sintaxe do XML.

I_VALIDATE não realiza validação em relação ao DTD ou ao XMLSchema.

I_VALIDATE faz parte da biblioteca **libmedia**

14.5 UDF BLOB Common Interface

Hoje em dia, multimídia é importante e útil para os usuários. O **DBMaker** oferece uma interface comum para acessar **BLOBs** usando um método de manipulação de arquivos, permitindo que os programadores escrevam facilmente **UDFs** para dados do tipo **BLOB**. **FILE**, **LONG VARCHAR** e **LONG VARBINARY** são os tipos de dados usados para armazenar dados **BLOB** em um banco de dados.

Muitas das novas funcionalidades no **DBMaker** exigem um **BLOB** temporário para processar resultados temporários. O **DBMaker** oferece suporte a **BLOBs** temporários para que os programadores possam escrever **UDFs** com mais facilidade. Um programador pode abrir um **BLOB** permanente, ler os dados, executar uma função de conversão ou algo semelhante, salvar o resultado em um novo **BLOB** temporário e retorná-lo em uma **UDF**. A **API** recupera este **BLOB** temporário como uma coluna **BLOB** normal.

BLOB Common Interface Functions

O **DBMaker** oferece funções de interface comum para **BLOBs** para que os programadores possam escrever **UDFs**. Um **DBA** deve definir o **DB_FoDir** no arquivo **dmconfig.ini** para o arquivo **BLOB** temporário antes de iniciar o banco de dados. Um **BLOB** temporário será criado em um arquivo externo no diretório **DB_FoDir**, com o formato de nome de arquivo "**__??????.TMP**", onde "?" representa um caractere de [0-9, A-Z]. Todos os nomes de arquivos que correspondem ao formato serão excluídos quando o banco de dados for desligado e reiniciado.

_UDFBBOPEN()

Abre um **BLOB** usando **bbObj** e retorna um manipulador através de **pHandle**. **bbObj** pode ser recuperado por **Arg[i]**, usando o **BLOB** com o argumento de entrada da **UDF**. A função retorna 0 se conseguir abrir o **BLOB** com sucesso; caso contrário, um código de erro será retornado.

```
nt    _UDFBbOpen(VAL    *Arg,    BBObj    bbObj,    i31    *pHandle);
```

_UDFBBREAD()

Esta função lê o **BLOB** que pertence ao manipulador especificado. Antes de chamar esta função, aloque um buffer (**pBuf**) com **szBuf** usando a função **_UDFAllocMem()** para obter os dados lidos. Os dados retornados serão armazenados em **pBuf** e o tamanho realmente lido estará em **szRead**. Se **szBuf** for não positivo, nenhum caractere será lido.

```
characters are read:
int _UDFBbRead(VAL *Arg, i31 handle, i31 szBuf, i31 *szRead, char
*pBuf);
```

_UDFBBSEEK()

Esta função é usada para definir a posição da próxima operação de saída em um **BLOB**. A nova posição é definida em **offset** bytes a partir do início, da posição atual ou do final do arquivo, de acordo com **ptrname**, usando os valores **SEEK_BB_BEG**, **SEEK_BB_CUR** ou **SEEK_BB_END** definidos em **libudf.h**.

A função só funciona entre o período de **_UDFBbOpen()** e **_UDFBbClose()**, mas não entre **_UDFBbCreate()** e **_UDFBbClose()**.

```
int _UDFBbSeek(VAL *Arg, i31 handle, i63 offset, i16 ptrname);
```

_UDFBBCUROFFSET

A função retorna a posição atual de um BLOB aberto ou o deslocamento em um BLOB por pOffset, mas retornará no máximo 2^{31} - 1, mesmo quando o deslocamento atual for maior ou igual a 2^{31}.

```
int _UDFBbCurOffset(VAL *Arg, i31 handle, i31 *pOff
```

_UDFBBCUROFFSETEX

Diferentemente de _UDFBbCurOffset, esta função sempre retorna a posição atual de um BLOB aberto ou o deslocamento em um BLOB por pOffset.

```
int _UDFBbCurOffsetEx(VAL *Arg, i31 handle, i63 *pOffset);
```

_UDFBBCLOSE()

Fecha o BLOB aberto por _UDFBbOpen() ou criado por _UDFBbCreate().

```
int _UDFBbClose(VAL *Arg, i31 handle);
```

_UDFBBCREATE()

Cria um BLOB temporário e retorna um handle para _UDFBbWrite(). O chamador deve preparar o espaço para a estrutura BBObj apontada por pBbObj e escrita por _UDFBbCreate(), _UDFBbWrite() e _UDFBbClose(). BBObj é usado para identificar este BLOB temporário. Por exemplo, se você quiser excluir o BLOB temporário chamado _UDFBbDrop() usando o argumento BBObj.

Se for bem-sucedido, pHandle retornará um handle de BLOB semelhante ao handle do arquivo aberto escrito por _UDFBbWrite() e fechado por _UDFBbClose(). Alternativamente, especifique o BLOB temporário a ser criado em arquivo (BB_TEMP_FO) ou em memória (BB_TEMP_MEM). Se o chamador especificar o BLOB temporário na memória, isso não significa que o BLOB temporário será criado na memória - uma limitação de memória pode impedir isso. Os BLOBs temporários na memória podem ser convertidos em arquivos pelo sistema operacional se os BLOBs temporários originais na memória ou os dados de entrada excederem o limite de tamanho. Os programadores não devem depender desse recurso ao codificar.

A função retorna 0 se for bem-sucedida e um código de erro será retornado caso contrário. Antes de ler o novo BLOB temporário, você deve fechá-lo usando _UDFBbClose() e, em seguida, reabri-lo usando _UDFBbOpen(). _UDFBbSeek() não pode ser usado em BLOBs temporários, a menos que eles sejam fechados e reabertos para leitura.

```
int _UDFBbCreate(VAL *Arg, BBObj *pBbObj, i31 *pHandle, i31 Opt);
```

_UDFBBWRITE()

Após usar _UDFBbCreate() para criar um BLOB temporário, escreva dados nele usando _UDFBbWrite(). O handle é obtido de _UDFBbCreate(), pBuf aponta para os dados de entrada e seu comprimento é szBuf. A função retorna 0 se for bem-sucedida, caso contrário, um código de erro será retornado.

```
int _UDFBbWrite(VAL *Arg, i31 handle, i31 szBuf, char *pBuf);
```

_UDFBBDROP()

Normalmente, você não deve excluir um BLOB temporário se ele for retornado de uma UDF; o sistema controlará seu ciclo de vida. Se você não retornar o BLOB criado, é melhor usar esta função para excluir o BLOB temporário. Esta função não pode ser usada em um BLOB permanente; fazer isso retornará o erro ERR_BLOB_INV_BLOB. A função retorna 0 se for bem-sucedida, caso contrário, um código de erro será retornado:

```
int _UDFBbDrop(VAL *Arg, BBObj bbObj);
```

_UDFBBSIZE()

Esta função retorna o tamanho dos dados de um BLOB através de `pLen`. `BbObj` pode ser um BLOB permanente ou um BLOB temporário. No entanto, ela retornará no máximo 2^31 - 1, mesmo que o tamanho seja maior ou igual a 2^31. A função retorna 0 se for bem-sucedida, caso contrário, um código de erro será retornado:

```
int _UDFBbSize(VAL *Arg, BBObj bbObj, i31 *pLen);
```

_UDFBBSIZEEX()

Diferentemente de `_UDFBbSize`, esta função sempre retorna o tamanho real dos dados de um BLOB através de `pLen`. `BbObj` pode ser um BLOB permanente ou um BLOB temporário. A função retorna 0 se for bem-sucedida; caso contrário, um código de erro será retornado:

```
int _UDFBbSizeEx(VAL *Arg, BBObj bbObj, i63 *pLen);
```

Exemplo

A seguir, é demonstrado como criar a função definida pelo usuário, MYCONVERT, com entrada no formato varchar e saída como um BLOB temporário.

 Para criar a função definida pelo usuário, MYCONVERT:

1. Construa uma biblioteca dinâmica no UNIX usando `myudf.c` (o código-fonte será apresentado posteriormente):

```
cc -g -c myudf.c
ld -G -o myudf.so myudf.o
```

2. Inicie o banco de dados
3. No prompt do dmSQL, digite:

```
dmSQL> CREATE FUNCTION myudf.myconvert(VARCHAR(100)) // input string
 2> RETURNS LONG VARCHAR; // output BLOB
dmSQL> SELECT myconvert(c1) FROM mytable; // output temp BLOB
```

O código-fonte para a UDF MYCONVERT pode ser algo assim:

```
#include "libudf.h"
int MYCONVERT(int nArg, VAL args[])
{
 int rc = 0, trc; /* return code */
 BBObj tmpobj; /* output temp BLOB */
 i31 handle; /* handle of created temp BLOB */
 boolean fgCreate = FALSE; /* temp BLOB has been created? */
 char *pInData, pOutData[4096];/* input/output data buffer */
 i31 nInData, nOutData; /* input/output data buffer length */
 if (args[0].type == NULL_TYP)
 goto cleanup;
 pInData = args[0].u.xval; /* get input data */
 nInData = args[0].len; /* input data length */
 /* create a temp BLOB in file */
 if (rc = _UDFBbCreate(args, &tmpobj, &handle, BB_TEMP_FO))
 goto cleanup;
 fgCreate = TRUE;
/* any real processing function */
 RealConvert(pInData, nInData, pOutData, &nOutData);
 /* write result data to temp BLOB */
 if (rc = _UDFBbWrite(args, handle, nOutData, pOutData))
 goto cleanup;
 /* close created temp BLOB ( temp BLOB is still alive) */
 if (rc = _UDFBbClose(args, handle))
 goto cleanup;
 args[0].type = BLOB_TYP;
```

```
 args[0].len = BBID_SIZE;
 args[0].u.xval = (char *)&tmpobj;
 /* _RetVal() does a copy from this local buffer */
cleanup:
 if (rc)
 {
 /* error handle */
 if (fgCreate)
 {
 _UDFBbClose(args, handle); /* close created temp BLOB */
 trc = _UDFBbDrop(args, tmpobj); /* drop it because of failure */
 if (trc > rc)
 rc = trc;
 }
 return rc;
 }
else
 return _RetVal(args, args[0]);
}/* MYCONVERT() */
```

Troubleshooting Errors

Use o seguinte para solucionar erros ao escrever uma UDF BLOB usando a interface comum do BLOB.

ERRO (327): A COLUNA BLOB AINDA NÃO FOI ABERTA OU CRIADA A função deve usar _UDFBbOpen() para abrir o BLOB ou _UDFBbCreate() para criar um novo BLOB temporário antes de usar outras interfaces de função BLOB.

ERRO (328): O DESLOCAMENTO DA COLUNA BLOB É INVÁLIDO Quando uma UDF usando _UDFBbSeek() procura um deslocamento maior que o comprimento do BLOB.

ERRO (331): ESTE BLOB NÃO ESTAVA NO ESTADO DE CRIAÇÃO _UDFBbWrite() só pode funcionar em um BLOB temporário criado por _UDFBbCreate() e não deve estar fechado. Por exemplo, se você usá-lo em um BLOB aberto por _UDFBbOpen(), este erro ocorrerá.

ERRO (330): ESTE BLOB NÃO ESTAVA NO ESTADO DE ABERTURA _UDFBbRead() só pode funcionar em um BLOB (incluindo BLOBs temporários) aberto por _UDFBbOpen().

ERRO (332): O OBJETO BLOB AINDA NÃO ESTÁ FECHADO Sempre que _UDFBbOpen() ou _UDFBbCreate() forem usados para abrir um BLOB, os programadores devem chamar _UDFBbClose() para fechar o BLOB aberto.

ERRO (322): NÃO HÁ DIRETÓRIO DE OBJETO DE ARQUIVO NO ARQUIVO DE CONFIGURAÇÃO; NÃO É POSSÍVEL INSERIR OBJETO DE ARQUIVO Se BLOBs temporários forem usados, a palavra-chave DB_FoDir no arquivo dmconfig.ini deve ser configurada. Se não estiver configurada, tentar criar um BLOB temporário pode falhar e este erro ocorrerá.

14.6 UDF related dmconfig.ini keywords

DB_StrSz

Além de DB_LbDir e DB_FoDir, há também uma palavra-chave relacionada, DB_StrSz, no arquivo dmconfig.ini:

```
DB_StrSz=<value>
```

Esta palavra-chave indica o comprimento dos dados retornados do tipo STRING, usados apenas por funções definidas pelo usuário (UDF). Como as UDFs só podem retornar dados de tamanho fixo, essas palavras-chave podem limitar o tamanho dos dados STRING, para evitar o recebimento de strings muito longas. O valor padrão é 255, e o intervalo válido é de 1 a 4.096. Pode ser usado em um cliente ou servidor, sendo que o cliente tem prioridade mais alta.