Reference Manual

The Multilevel Grid File (MLGF)

Version 4.0
Manual Release 1

July 2016

Advanced Information Technology Research Center (AITrc)

KAIST

1. Introduction

MLGF 4.0 is an implementation of the Multilevel Grid File, which is a dynamic hashed file organization. The Multilevel Grid File (MLGF) dynamically modifies its structure by means of splitting and merging. Splitting a page occurs when a new record being inserted into a page causes an overflow, while merging a page with other page occurs when a record being deleted from the page causes underflow.

MLGF 4.0 has following features:

- Creating an MLGF
- Opening an MLGF for reading or writing
- Closing an opened MLGF
- Inserting a record into the MLGF
- Deleting a record from the MLGF
- Retrieving records in a given region from the MLGF

2. Application Programming Interface

MLGF 4.0 provides 14 functions: MLGF_Init(), MLGF_Final(), MLGF_CreateIndex(), MLGF_OpenIndex(), MLGF_CloseIndex(), MLGF_InsertObject(), MLGF_DeleteObject(), MLGF_Query(), MLGF_Dump(), etc. This section explains each function in detail.

MLGF 4.0 uses the following data type definitions to improve its portability.

- For 32bit OS,
```
typedef long            Four;
typedef unsigned long   UFour;
typedef short           Two;
typedef char            One;
```
- For 64bit OS,
```
typedef int             Four;
typedef unsigned int    UFour;
typedef short           Two;
typedef char            One;
```

Here, the data type One has the size of one-byte. Others are defined likewise.

2.1. MLGF_Init

```
Four MLGF_Init( nBuffers)
Four     nBuffers;          /* IN      number of buffers */
```

MLGF_Init() allocates and initializes memory spaces for data structures of the buffer manager used by an MLGF. ‘nBuffers’ is the number of buffers to be used. To use the MLGF compiled to use the buffer manager, the memory spaces must be allocated and be initialized by using MLGF_Init(). Finishing the use of the MLGF, the memory spaces must be freed by using MLGF_Final().

Example

The following example is a part of the program that allocates and initializes, and that frees the memory spaces for data structures related to the buffer manager used by the MLGF.

```
e = MLGF_Init(400);
if ( e < eNOERROR ) {
        /* error handling routine */
}

…


e = MLGF_Final();
if ( e < eNOERROR ) {
        /* error handling routine */
}
```

2.2. MLGF_Final

```
Four MLGF_Final()
```

MLGF_Final() frees the memory spaces for data structures of the buffer manager used by an MLGF. Finishing the use of the MLGF compiled to use the buffer manager, the memory spaces must be freed by using MLGF_Final().

Example

See the example of MLGF_Init()

2.3. MLGF_CreateIndex

```
Four MLGF_CreateIndex( path, numOfKeys, numOfAttrs, attributeType, minMaxTypeVector)
One              *path;              /* IN    full name of the file */
One              numOfKeys;                   /* IN    number of keys */
One              numOfAttrs;         /* IN    number of attributes */
AttributeType    *attributeType;     /* IN    types of each attribute */
MLGF_HashValue   minMaxTypeVector;   /* IN    bit vector of flags indicating types of
                                              each key's extreme hash value */
One              useAdditionalFunc   /* IN    flag of additional functionalities use */
```

MLGF_CreateIndex() creates a new MLGF named with pathname indicated by the parameter ‘path.’ When MLGF_CreateIndex() creates an MLGF, a schema information must be specified about records to be stored in the file. The parameter ‘numOfAttrs’ specifies the number of attributes in the record including the key attributes, and the parameter ‘attributeType’ specifies a data types for each attribute. The parameter ‘numOfKeys’ specifies the number of keys used in the MLGF.

The parameter ‘attributeType’ specifies the types of attributes. Here, key attributes precede other ones.

```
typedef struct {
        enum { INT, FLOAT, STRING, VARSTRING } dataType;     /* data type */
        Two                                    length;       /* length */
} AttributeType;
```

The parameter ‘dataType’ can have one of the following four: INT, FLOAT, STRING, and VARSTRING. The parameter ‘length’ represents the size of an attribute. If the parameter ‘dataType’ is INT or FLOAT, the parameter ‘length’ has the size of the int type or the float type. If the parameter ‘dataType’ is STRING, the parameter ‘length’ has the size of a string. And in the case of VARSTRING, the maximum size of a string is assigned to the parameter ‘length.’

The parameter ‘minMaxTypeVector’ specifies the types of an extreme hash value applied to the keys. We can improve the MLGF range query performance by assigning appropriate values to the parameter ‘minMaxTypeVector.’ There are two types of extreme hash value available: MINTYPE and MAXTYPE.

The parameter ‘useAdditionalFunc’ specifies whether to use additional functionality or not(TRUE or FALSE). Additional functionality supported in MLGF 4.0 is as follows. 1. allow data records of variable lengths, 2. allow data records with duplicate hash values of organizing attributes.

Example

The following example creates an MLGF whose record has three attributes that are INT, FLOAT, VARSTRING. Among them, the first two attributes are key attributes. The created MLGF has the name ‘test.mlgf.’

```
AttributeType      attrType[3];                    /* variable declaration */
MLGF_HashValue     minMaxTypeVector;
Four               e;                              /* error code */

attrType[0].dataType = INT;
attrType[0].length = sizeof(int);                  /* first attribute */
attrType[1].dataType = FLOAT;
```

```
attrType[1].length = sizeof(float);                  /* second attribute */
attrType[2].dataType = VARSTRING;
attrType[2].length = 20;                             /* third attribute */

MLGF_KEYDESC_SET_MINTYPE(minMaxTypeVector, 0);       /* first key's extreme hash
                                                        type */
MLGF_KEYDESC_SET_MAXTYPE(minMaxTypeVector, 1);       /* second key's extreme hash
                                                        type */

/* create mlgf file */
e = MLGF_CreateIndex( "test.mlgf", 2, 3, attrType, minMaxTypeVector, TRUE);
if ( e < eNOERROR ) {
        /* error handling routine */
}
```

2.4. MLGF_OpenIndex

```
Four MLGF_OpenIndex( path, mlgfd)
One     *path;    /* IN     path */
Two     *mlgfd;   /* OUT    mlgf file descriptor */
```

MLGF_OpenIndex() opens a file with the pathname indicated by the parameter ‘path’ and returns a file descriptor in the parameter ‘mlgfd.’ To access an MLGF, the MLGF must be opened by using MLGF_OpenIndex(). Finishing the use of the MLGF, it must be closed by using MLGF_CloseIndex().

Example

The following example is a part of the program that opens and closes the MLGF “test.mlgf.”

```
Two     mlgfd;    /* mlgf file descriptor */
Four    e;        /* error code */

e = MLGF_OpenIndex( "test.mlgf", &mlgfd);
if ( e < eNOERROR ) {
        /* error handling routine */
}

…


e = MLGF_CloseIndex( mlgfd);
if ( e < eNOERROR ) {
        /* error handling routine */
}
```

2.5. MLGF_CloseIndex

```
Four MLGF_CloseIndex( mlgfd)
Two        mlgfd;    /* IN      mlgf file descriptor */
```

MLGF_CloseIndex() closes the MLGF indicated by ‘mlgfd.’ ‘mlgfd’ is the file descriptor returned by MLGF_OpenIndex(). After using the MLGF, it must be closed by using MLGF_CloseIndex().

This function writes all updated information related with MLGF to disk and closes it

Example

See the example of MLGF_OpenIndex()

2.6. MLGF_InsertObject

```
Four MLGF_InsertObject( mlgfd, record)
Two                mlgfd;            /* IN      mlgf file descriptor */
AttributeHeader    *attrValues;      /* IN      record array */
```

MLGF_InsertObject() inserts a record into the MLGF indicated by the parameter ‘mlgfd.’ The parameter ‘mlgfd’ is the file descriptor returned by MLGF_OpenIndex(). The record’s value to be inserted are specified in the form of an array of attributes. The parameter ‘attrValues’ is a pointer pointing to that array. AttributeHeader used for specifying values of attributes is defined as flows.

```
typedef struct {
          Two       length;             /* length */
          union {
                    int       *intPtr;  /*pointer */
                    char      *charPtr;
                    float     *floatPtr;
          } field;
} AttributeHeader;
```

The parameter ‘length’ represents the length of an attribute. The parameter ‘field’ is the union of three pointers. An appropriate pointer among these three pointers can point to the value of a particular attribute. For example, if an attribute’s data type is INT, ‘intPtr’ is used for accessing the data. In the cases of STRING or VARSTRING, ‘charPTR’ is used for accessing the data.

Example

When a record (10, 10.1, “Hello”) is inserted into the MLGF created in chapter 2.3, the following steps are required.

```
AttributeHeader   attr[3];
Four      e;          /* error code */

…

/* allocate memory */
attr[0].length = sizeof(int);              /* first attribute */
attr[0].field.intPtr = (int *)malloc( sizeof( int));
attr[1].length = sizeof(float);            /* second attribute */
attr[1].field.floatPtr = (float *)malloc( sizeof( float));
attr[2].length = 20;                       /* third attribute */
attr[2].field.charPtr = (char *)malloc( sizeof( char) * 20);

/* assign values */
*(attr[0].field.intPtr) = 10;              /* first attribute */
*(attr[1].field.floatPtr) = 10.1;          /* second attribute */
attr[2].length = strlen(“Hello”) + 1;      /* third attribute */
strcpy( attr[2].field.charPtr, “Hello”);

/* insert record */
e = MLGF_InsertObject( mlgfd, attr);
if ( e < eNOERROR ) {
          /* error handling routine */
}

…

free( attr[0].field.intPtr);
free( attr[1].field.floatPtr);
free( attr[2].field.charPtr);
```

2.7. MLGF_DeleteObject

```
Four MLGF_DeleteObject( mlgfd, record)
Two               mlgfd;          /* IN     mlgf file descriptor */
AttributeHeader   *attrValues;    /* IN     record to delete */
```

MLGF_DeleteObject() deletes the record that ‘attrValues’ specifies, from the MLGF that ‘mlgfd’ specifies. ‘mlgfd’ is the file descriptor returned by MLGF_OpenIndex(). The parameter ‘mlgfd’ is the file descriptor and the parameter ‘attrValues’ is the attribute header used for finding the records to be deleted.

Example

For examples, since the key value of the record (10, 10.1, “Hello”) used in Section 2.6 is (10, 10.1), it can be deleted as follows.

```
AttributeHeader   attr[2];
Four       e;            /* error code */

…

/* allocate memory */
attr[0].length = sizeof(int);              /* first attribute */
attr[0].field.intPtr = (int *)malloc( sizeof( int));
attr[1].length = sizeof(float);            /* second attribute */
attr[1].field.floatPtr = (float *)malloc( sizeof( float));
attr[2].length = 20;                       /* third attribute */
attr[2].field.charPtr = (char *)malloc( sizeof( char) * 20);

/* assign values */
*(attr[0].field.intPtr) = 10;              /* first attribute */
*(attr[1].field.floatPtr) = 10.1;          /* second attribute */
attr[2].length = strlen(“Hello”) + 1;      /* third attribute */
strcpy( attr[2].field.charPtr, “Hello”);

/* delete record */
e = MLGF_DeleteObject( mlgfd, attr);
if ( e < eNOERROR ) {
          /* error handling routine */
}
```

2.8. MLGF_Query

```
Four MLGF_Query( mlgfd, region, num, records)
Two               mlgfd;          /* IN     mlgf file descriptor */
QueryRegion       *region;        /* IN     query region */
Four              num;            /* IN     buffer size */
AttributeHeader   *records;       /* OUT    buffer to return the records */
```

MLGF_Query() returns all the records that belong to the given query region. The parameter ‘mlgfd’ is the file descriptor returned by MLGF_OpenIndex(). The parameter ‘region’ representing the query region is defined by giving the maximum and the minimum values of each key attribute. QueryRegion used for specifying the maximum and minimum values of a key attribute is defined as follows.

```
typedef struct {
          One               fullDomainFlag;   /* TRUE if region is full domain */
          AttributeHeader   minkey;           /* minimum key value */
          AttributeHeader   maxkey;           /* maximum key value */
} QueryRegion;
```

‘fullDomainFlag’ represents whether the query region covers the whole domain or a part of it. If the query region covers the whole domain, ‘fullDomainFlag’ is ‘TRUE,’ and if the query region covers a part of the domain, ‘fullDomainFlag’ is ‘FALSE.’ ‘minKey’ and ‘maxKey’ represent minimum and maximum values of the query region. They are used only in the case of a partial domain query.

All the records that belong to the query region are returned in the parameter ‘records.’ ‘records’ is a pointer to the array of type AttributeHeader. The y-th attribute value of the x-th record is returned as the (x * # of attributes + y)-th element in the array.

This function returns the number of the records that belong to the query region. If the number of records is greater than ‘num’, only ‘num’ records are copied to ‘records,’ and this function returns eTOOMANYRECORDS. If ‘records’ has NULL, no record is copied only returning the number of the records that belong to the query.

Example

For the MLGF that was created in Section 2.3, the following example finds all the records whose first key attribute has the value between 5 to 15 and whose second key attribute has the value over the entire domain.

```
QueryRegion       region[2];
AttributeHeader   *attr;
int               nRecords;
int               minKeyValue, maxKeyValue;
int               *intArray;
float             *floatArray;
char              (*stringArray)[20];
Four              e;
int               i, num;


/* set region */
region[0].fullDomainFlag = FALSE;   /* partial domain */
minKeyValue = 5;
region[0].minKey.length = sizeof(int);
region[0].minKey.field.intPtr = &minKeyValue;
maxKeyValue = 15;
region[0].maxKey.length = sizeof(int);
region[0].maxKey.field.intPtr = &maxKeyValue;
region[1].fullDomainFlag = TRUE;    /* full domain */

/*get the number of records in the region */
nRecords = MLGF_Query( mlgfd, region, 0, NULL);
if ( nRecords < eNOERROR ) {
        /* error handling routine */
}

/* allocate memory */
num = nRecords * 3;          /* 3 is the number of attributes */
attr = (AttributeHeader*)malloc(sizeof(AttributeHeader)*num);
intArray = (int *)malloc(sizeof(int)*nRecords);
```

```
floatArray = (float *)malloc(sizeof(float)*nRecords);
/* 20 is the maximum length of the third attribute */
stringArray = (char(*)[20])malloc(sizeof(char)*nRecords*20);
for ( i = 0; i < nRecords; i++) {
        attr[i*3+0].field.intPtr = &intArray[i];
        attr[i*3+1]. field.floatPtr = &floatArray[i];
        attr[i*3+2]. field.charPtr = &stringArray[i][0];
}

/* find records */
e = MLGF_Query( mlgfd, region, nRecords, attr);
if ( e < eNOERROR ) {
        /* error handling routine */
}
```

2.9. MLGF_Dump

```
Four MLGF_Dump( mlgfd, recordOnly, outputfd)
Two     mlgfd;          /* IN   mlgf file descriptor */
Boolean recordOnly;     /* IN   flag that indicates what to print out */
FILE    *fd;            /* OUT  output file descriptor */
```

MLGF_Dump() prints out all the records or the tree structure of the MLGF to the output file specified by ‘fd.’ If ‘recordOnly’ is TRUE (1), it prints out all the records, otherwise (FALSE (0)), it prints out the tree structure of the MLGF using preorder search.

The MLGF includes directory pages, leaf pages, and overflow pages. A directory page is a non-leaf node of an MLGF and has entries pointing to the next level directory pages or data pages. A leaf page is a leaf node of an MLGF and contains data records. An overflow page contains the records having identical keys when they occupy more than one third of a leaf page.

Example

This example shows the result of MLGF_Dump() when it is applied to the MLGF (“test.mlgf”) in Figure 1.The MLGF has three attributes (two of type INT and one of type FLOAT).

In this MLGF, the root page has two entries and each of them points the leaf pages 10 and 14 respectively. The leaf page 10 has three entries; the 0th entry has two records having the same key value. The leaf page 14 has three entries. The 0th entry has three records having the same key value, thus, these three records are stored in an overflow page.

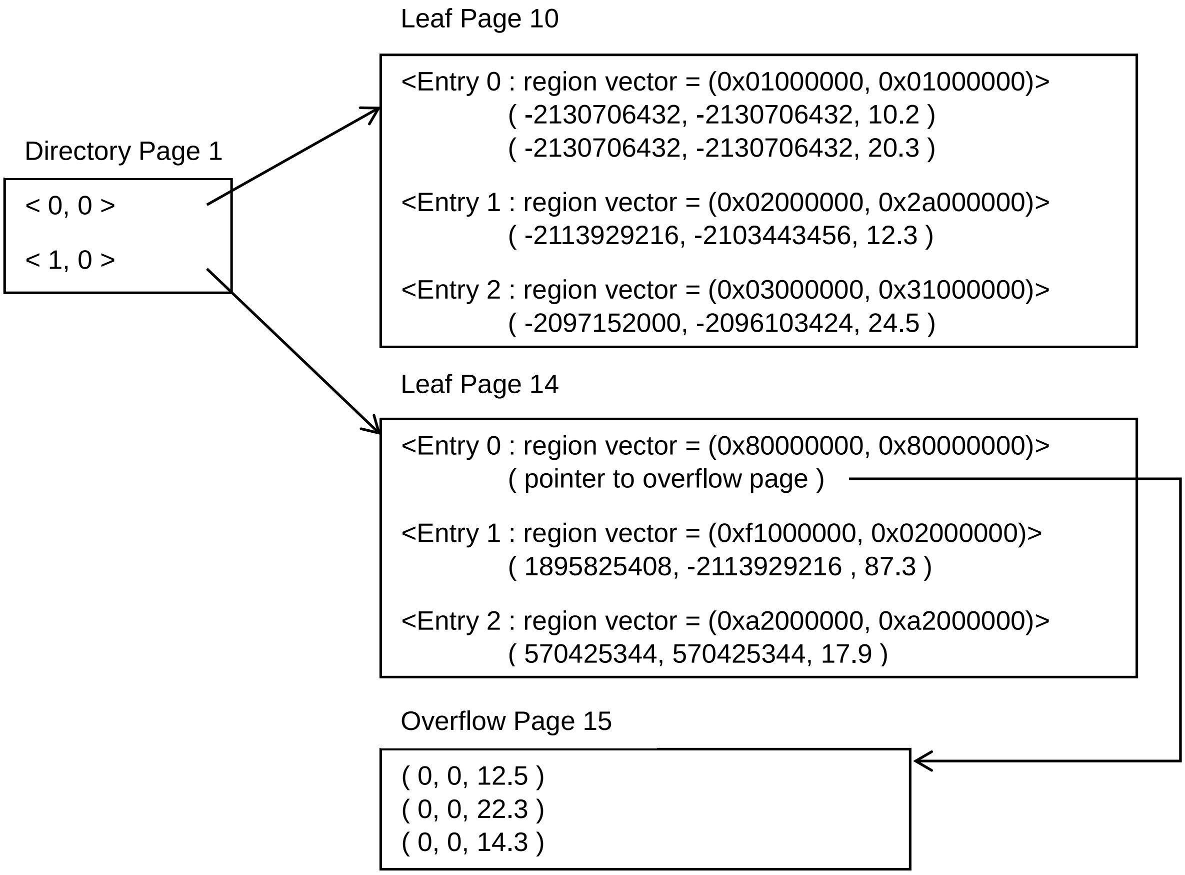
Leaf Page 10
<Entry 0 : region vector = (0x01000000, 0x01000000)>
(-2130706432, -2130706432, 10.2)
(-2130706432, -2130706432, 20.3)
<Entry 1 : region vector = (0x02000000, 0x2a000000)>
(-2113929216, -2103443456, 12.3)
<Entry 2 : region vector = (0x03000000, 0x31000000)>
(-2097152000, -2096103424, 24.5)
Directory Page 1
< 0, 0 >
< 1, 0 >
Leaf Page 14
<Entry 0 : region vector = (0x80000000, 0x80000000)>
(pointer to overflow page)
<Entry 1 : region vector = (0xf1000000, 0x02000000)>
(1895825408, -2113929216 , 87.3)
<Entry 2 : region vector = (0xa2000000, 0xa2000000)>
(570425344, 570425344, 17.9)
Overflow Page 15
(0, 0, 12.5)
(0, 0, 22.3)
(0, 0, 14.3)

그림 1: Tree structure of “test.mlgf”

```
Two     mlgfd;   /* mlgf file descriptor */
Four    e;       /* error code */

e = MLGF_OpenIndex( “test.mlgf”, &mlgfd);   /* open MLGF file */
if ( e < eNOERROR ) {
        /* error handling routine */
}

…

e = MLGF_Dump( mlgfd, FALSE, stdout);       /* dump MLGF file to screen */
if ( e < eNOERROR ) {
        /* error handling routine */
}

…

e = MLGF_CloseIndex( mlgfd);                        /* close MLGF file */
```

```
        if ( e < eNOERROR ) {
                /* error handling routine */
        }
```

Result of the MLGFdump:

```
 <depth : 1> Directory Page : 1
----------------------------------------------------
|    directory page dump
----------------------------------------------------
|    # of entries              : 2
|   Entry 0 |                  Page Ptr 10 |
|        0-th Hash Value|      valid bit   1|     hash value 01000000
|        1-th Hash Value|      valid bit   1|     hash value 01000000
|   Entry 1 |                  Page Ptr 14 |
|        0-th Hash Value|      valid bit   1|     hash value 80000000
|        1-th Hash Value|      valid bit   1|     hash value 80000000

 <depth : leaf> Leaf Page : 10
 Dump leaf page : total 3 entries
Entry 0 : region vector = <0x01000000, 0x01000000>, nRecords 2
   Record 0 : ( -2130706432, -2130706432, 10.2 )
   Record 1 : ( -2130706432, -2130706432, 20.3 )
Entry 1 : region vector = <0x02000000, 0x2a000000>, nRecords 1
   Record 0 : ( -2113929216, -2103443456, 12.3 )
Entry 2 : region vector = <0x03000000, 0x31000000>, nRecords 1
   Record 0 : ( -2097152000, -2096103424, 24.5 )

 <depth : leaf> Leaf Page : 14
 Dump leaf page : total 3 entries
Entry 0 : region vector = <0x80000000, 0x80000000>, Overflow Page Ptr 15

 <Overflow Page> Overflow Page : 15
 Dump overflow page : total 3 records
   Record 0 : ( 0, 0, 12.5 )
   Record 1 : ( 0, 0, 22.3 )
   Record 2 : ( 0, 0, 14.3 )

Entry 1 : region vector = <0xf1000000, 0x02000000>, nRecords 1
   Record 0 : ( 1895825408, -2113929216 , 87.3 )
Entry 2 : region vector = <0xa2000000, 0xa2000000>, nRecords 1
   Record 0 : ( 570425344, 570425344, 17.9 )
```

2.10. MLGF_Error

```
char *MLGF_Error( errorCode)
Four      errorCode;          /* IN      error code */
```

MLGF_Error() returns the error message corresponding to ‘errorCode.’ All functions in MGF 4.0 return an error code so that a user may confirm correct execution. This chapter explains the usage of MLGF_Error(). Section 2.13 explains error codes in detail.

Example

The following is the portion of the program that opens the MLGF “test.mlgf.”

```
Two     mlgfd;   /* mlgf file descriptor */
Four    e;       /* error code */

e = MLGF_OpenIndex( “test.mlgf”, &mlgfd);
if ( e < eNOERROR ) {
        /* error handling routine */
        printf(“MLGF_OpenIndex: (error %d) %s\n”, e, MLGF_Error( e));
        exit(1);
}
```

2.11. MLGF_GetN_KeyAttribute, MLGF_GetN_Attrribute, MLGF_GetAttributeType, MLGF_GetAttributeLength

These four functions allow users to reference the schema information of the record that stored in the MLGF. MLGF_GetN_KeyAttribute() and MLGF_GetN_Attribute() return the number of key attributes and the number of all the attributes of the record. MLGF_GetAttributeType() and MLGF_GetAttributeLength() return the type and the length of the record. ‘attributeNum’ is the number of the attribute to be referenced.

Example

The following example shows the schema information of the record that stored in the MLGF created in chapter 2.3.

```
Two             mlgfd;          /* mlgf file descriptor */
One             nAttributes;    /* number of attributes */
int             i;

…

nAttributes = MLGF_GetN_Attribute(mlgfd);

for( i = 0; I < nAttributes; i++) {
        printf(“Type of attribute #%d: %d\n”, i, MLGF_GetAttributeType(mlgfd, i);
        printf(“Length of attribute #%d: %d\n”, i, MLGF_GetAttributeLength(mlgfd, i);
)

…
```

2.12. Error Code

The API functions in MLGF 4.0 return the following error codes. They are stored in the file ‘mlgf.h’

```
#define eNOERROR                    0       /* no error */
#define eBADPARAMETER               -1
#define eDUPLICATEDRECORD           -2
#define eNOTFOUND                   -3
#define eTOOMANYRECORDS             -4
#define eMEMORYALLOCERR             -5
#define eSYSERR                     -6
#define eDUPLICATEDKEY              -7
#define eVARIABLELENGTHRECORD       -8
```

The errors that these error represent are as follows.

- eNOERROR

  The operation has been completed with no errors.

- eBADPARAMETER

  Parameters for the function have invalid values.

- eDUPLICATEDRECORD

  This error is returned when all attributes of a newly inserted record are identical to those of a previously inserted record. Only key attributes of newly inserted record are allowed to be the same with those of a previously inserted record.

- eNOTFOUND

  If MLGF_DeleteObject() can’t find the record specified, eNOTFOUND is returned.

- eTOOMANYRECORDS

  When the number of records found by MLGF_Query() is greater than the size of the buffer allocated, eTOOMANYRECORDS is returned.

- eMEMORYALLOCERR

  Allocating dynamic memory has failed.

- eSYSERR

  A system call has failed.

- eDUPLICATEDKEY

  When using MLGF that does not use additional functionality, this error is returned when the hash values of organizing attributes of a newly inserted record are identical to those of a previously inserted record.

- eVARIABLELENGTHRECORD

  When using MLGF that does not use additional functionality, this error is returned when there is a try to use variable length records.