# Minibase Short Tutorial

## Introduction

The Minibase DBMS has been implemented in C++ following the object oriented paradigm.

The whole system consists of four subsystems: the *Catalog*, where metadata information is kept, the *Storage Manager*, which takes care of the page allocation and deallocation, the *Buffer Manager*, which takes care of the page fetching in memory and the formers' flushing out to disk and the *Query Processing System*, which implements some relational operators. Each subsystem communicates with the other subsystems thought the public methods that each of the contained classes define. Each subsystem can be decomposed into further subsystems. For the purposes of the current project we will deal mainly with the storage manager as well as the buffer manager. Moreover, we will ignore the catalog completely.

The objects that implement the storage manager and the buffer manager are defined as global variables and can be accessed (as well as their methods) using the *MINIBASE_DB* and *MINIBASE_BM* macros respectively.

Before going on to explain the functionality that each subsystem provides in detail we will explain the error protocol that is used to propagate errors between subsystems and how the latter should be utilized.

## Error Protocol

Every subsystem creates error messages that describe the possible errors that will result. When an error is detected by a subsystem for the first time, that subsystem adds an error message to the global queue. The subsystem that discovered the error then returns a status code that indicates what subsystem it is: it identifies itself to the caller by returning its own ID. If the caller cannot recover from the error, the caller must then append a new error to the global queue. In this case, however, the thing appended to the queue is not a new message, but simply a pair of subsystem IDs: the original subsystem's ID, and the calling subsystem's ID.

Here is a possible (hypothetical) example of the errors that will be logged in the global error object:

```
DBMGR     "File Not Open"
BUFMGR    DBMGR
BTREE     BUFMGR
JOINS     BTREE
PLANNER   JOINS
FRONTEND  PLANNER
```

Note that there is some redundancy here: the recipient of an error is the source of an error in the next level. This helps ensure that the protocol is being followed properly.

The current protocol has all errors entered into a given variable of type *Status*. This variable is checked. If *OK*, then proceed. If *!OK*, then call *global_error::add_error(...)*. All errors must then be returned to the caller.

Problem: destructors are unable to set a non-global variable to a value. They can call *global_error::add_error()*, but there is no way for them to communicate failures.

## Error Numbers

When a subsystem posts a first error, it provides an error number that is specific to that subsystem. Each subsystem has its own set of error numbers, independent of the other subsystems. These numbers may become part of the subsystem's public interface (it is permissible to advertise what error numbers will be used in what circumstances, so that callers can recognize different conditions and handle the errors accordingly.

### Implementation

The simplest approach for declaring error numbers is to provide an enumeration. For example, here is the start of the buffer manager's enumeration of errors:

```
enum bufErrCodes { HASHTBLERROR,
                   HASHNOTFOUND,
                   BUFFEREXCEEDED, ... };
```

## Error Messages

Corresponding to its set of error numbers, each subsystem also must declare an array of error messages, and must make these messages available to the global error object. The index into the array must match the number of the corresponding error.

### Implementation

Here is an excerpt from the buffer manager's array of error messages, corresponding to the above enumeration:

```
static const char* bufErrMsgs[] = { "hash table error",
                                    "hash entry not found",
                                    "buffer pool full", ... };
```

This array of strings is declared *static*. So how can the buffer manager make these strings available to the global error object? By creating a static *error_string_table* object. The constructor of this object registers the error messages with the system when the program first starts.

Here is the buffer manager's *error_string_table* declaration, in the same file as the above array of strings:

*static error_string_table bufTable( BUFMGR, bufErrMsgs );*

## Posting Errors

There are three macros defined that make it easy to add errors when one discovers them. These macros also add the name of the file and the line number where the error happens, as a debugging aid.

- To add a **first** error, one may use the *MINIBASE_FIRST_ERROR* macro. For example, if the buffer manager detects that the buffer pool is too full to complete an operation, it posts its *BUFFEREXCEEDED* error like this:

  *MINIBASE_FIRST_ERROR( BUFMGR, BUFFEREXCEEDED );*

- To add a **chained** error, one may use the *MINIBASE_CHAIN_ERROR* macro. For example, if the buffer manager calls on the database manager to write a page to disk, and that operation fails, the buffer manager adds an error that records the fact that the execution path that failed went through it:

```
Status status = MINIBASE_DB->write_page( ... );
 if (status != OK )
     return MINIBASE_CHAIN_ERROR( BUFMGR, status );
```

- Sometimes, one wishes to post a different error message, but still acknowledge that the error resulted from a prior error. This is a combination of the above situations. For this, one may use the *MINIBASE_RESULTING_ERROR* macro:

```
Status status = MINIBASE_DB->write_page( ... );
if (status != OK )
    return MINIBASE_RESULTING_ERROR(BUFMGR,
                                    status, BUFFEREXCEEDED );
```

## Handling Errors

There are a number of ways to find out what has gone wrong in the system. The most primitive way is to print the global error object:

*minibase_errors.show_errors();*

If one needs the program to detect whether an error has happened, one may ask the error object if it has accumulated any errors:

```
if ( minibase_errors.error() )
      ...;
```

One may find out the subsystem that posted the first error:

```
if ( minibase_errors.originator() == BUFMGR )
 ...;
```

One may find out the error number of the first error:

```
if ( minibase_errors.error_index() == BUFFEREXCEEDED )
...;
```

If you wish to examine the entire set of errors that have been posted, you may use the error() method shown above to get a pointer to the first error record in the list; you may then use methods of the error_node class to get details of the error, and to traverse the entire list of errors.

## Status values

There are four groups of status codes:

- OK. This is the normal return status. It is in a class by itself.
- The layer, or subsystem ID, status codes (BUFMGR, RECOVERYMGR, LOGMGR, SHAREDMEMORYMGR, BTREE, SORTEDPAGE, BTINDEXPAGE, BTLEAFPAGE, LINEARHASH, GRIDFILE, RTREE, JOINS, PLANNER, PARSER, OPTIMIZER, FRONTEND, CATALOG, DBMGR, RAWFILE, LOCKMGR, XACTMGR, HEAPFILE, SCAN). A subsystem notifies its caller of an error by adding an error to the global error list and returning its subsystem ID status code. The caller may then inspect the error and decide how to handle it.
- DONE and FAIL. DONE is a special code for non-errors that are nonetheless not OK, it generally means finished or not found. FAIL is for errors that happen outside the bounds of a subsystem.
- The last category is a set of deprecated status codes that refer to specific problems. These are being replaced by the subsystem and error index mechanism as time permits.

# Storage Manager

Each database is basically a UNIX file and consists of several relations viewed as heapfiles and their indexes within it.

The storage manager creates and manipulates the database.

It keeps two basic kinds of global information. The first is a map of which database pages have been allocated. The second is a directory of files created within the database. The first page of the database (page ID 0) is reserved for a special structure that holds global information about the database, like the number of pages in the database. After this special first-page information comes the first (of possibly many) *directory page*. A directory page is where the storage manager keeps track of the files created within the

database. The second page (page ID 1), and as many subsequent pages as needed, holds the *space map*, which is a bitmap representing pages allocated in the database.

You are encouraged to look at the storage manager's class declarations and definitions located in files *db.h* and *db.C* respectively.

In what follows we are going to describe the functionality of the methods most probably needed for the current project. For additional methods, as well as details you are referred to the above two files.

*Status allocate_page(PageId& start_page_num, int run_size = 1);*

Allocates a set of pages where the run size (i.e. the number of pages) is taken to be 1 by default. It returns the page number (into the *start_page* variable) of the first page of the allocated run.

**********

*Status deallocate_page(PageId start_page_num, int run_size = 1);*

Deallocates a set of pages starting at the specified page number and a run size can be specified.

**********

*Status read_page(PageId pageno, Page* pageptr);*

Reads the contents of the specified page (i.e. *pageno*) into the given memory area (i.e. pageptr).

**********

*Status write_page(PageId pageno, Page* pageptr);*

Writes the contents of the specified page on disk.

Notes: The Page class is declared in *page.h* file. *PageId* is an alias for type int. It is declared in file *minirel.h*. Moreover, in the same file the type RID is defined, which is the unique record ID, which consists of a page number and a page offset.