

# *eXtremeDB*<sup>™</sup>

---

## Transaction Logging Addendum

(C) 2006-2007 McObject LLC  
22525 SE 64th Place, Suite 302  
Issaquah, WA 98027 USA

Tel: +1-425-888-8505  
Fax: +1-425-888-8508  
Email: [info@mcobject.com](mailto:info@mcobject.com)  
Web: <http://www.mcobject.com>

|  |    |
|--|----|
| Introduction.....                                  | 3  |
| Building the applications .....                    | 4  |
| Program Flow.....                                  | 4  |
| Transaction Log Files and Directory Structure..... | 5  |
| Transaction Logging API .....                      | 7  |
| mco_log_start.....                                 | 9  |
| mco_log_stop .....                                 | 10 |
| mco_log_recover.....                               | 11 |
| mco_log_save.....                                  | 12 |
| mco_log_delete_old .....                           | 13 |
| mco_log_deferred_deletion .....                    | 14 |
| mco_log_set_compression.....                       | 15 |
| DDL Requirements .....                             | 16 |

# Introduction

eXtremeDB-TL increases the options for persistence of eXtremeDB databases with the introduction of *transaction logging* – a process that journals changes made to a database (by transactions), as they are made. With transaction logging enabled, the eXtremeDB runtime captures database changes and writes them to a file known as a *transaction log*. In the event of a hardware or software failure, the eXtremeDB runtime can recover the database using the log. Logging is performed through periodic checkpoints, where the image of the database is saved to persistent storage, and all intermediate changes to the database are written to the log files.

Transaction logging does not alter the all in-memory architecture of eXtremeDB which retains a performance advantage over disk-based databases. Read performance is unaffected by transaction logging and write performance will far exceed write performance of traditional disk-based databases. The reason is simple to see: eXtremeDB transaction logging requires exactly one write to the file system for one database transaction. A disk-based database, however, will perform many writes per transaction (data pages, index pages, transaction log, etc) and the larger the transaction and the more indexes that are modified, the more writes that are necessary.

To minimize the performance impact, eXtremeDB equips developers with all the controls they need to tune their applications. Following its design principals, many eXtremeDB features are parameterized so that programmers can invoke the features most appropriate for their application scenario, whether their priority is maintaining the highest performance possible, or ensuring the highest level of transaction durability. For example, transaction logging may be turned on or off at runtime and, when turned on, logging may be set to different levels of transaction durability, allowing system designers to make intelligent trade offs between performance and risk for lost transactions.

Transaction Logging provides durability of eXtremeDB in-memory databases by providing the Transaction Log API. In the TL-enabled version of the eXtremeDB, every update action of a transaction is recorded in the in-memory buffers. When the transaction is committed these buffers are appended to the *database log files*. No records are added to the log if the transaction is read-only.

For recovery, the log is applied chronologically from the last checkpoint. A checkpoint is created periodically by the application when it requests a complete backup of the database. After the in-memory image is saved on the persistent media (a checkpoint), the transaction log created before the checkpoint is erased.

# Building the applications

In order to enable the transaction logging feature, in addition to the core eXtremeDB library, the application should be linked with the transaction log library *libmcolog.a* for UNIX platforms and (*mcolog.lib* for Windows platforms).

## Program Flow

The program flow of applications utilizing eXtremeDB Transaction Logging is as follows:

The application opens a database and gets a connection handle to it

```
mco_db_open()
```

```
mco_db_connect()
```

Transaction logging is started and transaction log internals are initialized

```
mco_log_start()
```

The application then recovers the database. In the normal course, the eXtremeDB-TL loads the last checkpoint image that was established during an orderly shutdown. In the case of an abnormal termination, transactions from one or more transaction log files will be recovered. If no checkpoint image is found, `MCO_E_LOG_IMG_NOT_FOUND` is returned. This would be a normal return the very first time a database is created, or could indicate media failure.

```
mco_log_recover()
```

Periodically, the application checkpoints the database

```
mco_log_save()
```

At the application's discretion, the transaction logging can be temporarily or permanently halted

```
mco_log_stop()
```

As the final steps of an orderly shutdown, a database checkpoint is created then the database is closed

```
mco_log_save()
```

```
mco_db_disconnect()
```

```
mco_db_close()
```

# Transaction Log Files and Directory Structure

As noted above, eXtremeDB transaction logging includes periodic checkpoints, where the image of the database is saved to persistent storage, and all intermediate changes to the database are written to the log files. The image file names have a **.img** extension and the transactions are written into the log files with the a **.log** extension. In order to increase the performance of transaction logging, eXtremeDB maintains a directory tree where the checkpoint files and the individual log files are kept. The number of files kept in each subdirectory and the maximum size of the log files are application-defined (see `mco_log_start()`). The root of the directory tree is passed as a parameter to the `mco_log_start()` API. The subdirectories under the root follow the naming convention ***Dnnnnnnn***, where *nnnnnnn* is a 7 digit number. The subdirectories are numbered sequentially starting from zero. When a checkpoint is requested via `mco_log_save()`, the runtime looks for an empty subdirectory in the tree. If one is not found, a subdirectory is created and the database image file **`mcodeb.img`** is written into it. After the new image file is successfully written, all old images and log files are deleted. An application may specify that old files are erased asynchronously. See `mco_log_deferred_deletion()`.

New log files are first created in the same directory as the image file. Log files are named ***Lnnnnnnn***, where *nnnnnnn* is a 7-digit number. When transaction logging is enabled via `mco_log_start()`, each transaction is recorded in the log file. When the size of the log file exceeds the maximum size (see `log_args_t.maxlogsize`), a new log file is created. Once the number of log files in the transaction logging current working directory has reached the maximum specified in `mco_log_start()` (see `log_args_t.maxfiles`), the runtime looks for an empty subdirectory. If there is no empty subdirectory in the tree, a new subdirectory is created. An index file, **`nextdir.idx`**, containing the catalog number is placed in the current directory. Note that the number of subdirectories in the log directory tree is not limited. The `log_args_t.maxfiles` value and frequency of `mco_log_save()` should be coordinated so that old log files are deleted frequently enough that the new directories are created rarely.

The following table depicts an example of what a directory tree might look like at a point in time, given a maxfiles value of '3'.

|           |  |
|-----------|--|
| Directory | ./D0000000                                   |
| File      | mcodb.img                                    |
| File      | L0000000                                     |
| File      | L0000001                                     |
| File      | L0000002                                     |
| File      | nextdir.idx (contains reference to D0000001) |
| Directory | ./D0000001                                   |
| File      | L0000000                                     |

When invoked, the `mco_log_recover()` function looks for the most recent image file and loads the database from it. It then reads the log files starting from the log file 0 (L0000000), and applies each transaction read from the log file to the in-memory database. After the current log file is processed, the function looks for the next log file and repeats the process. After the last log file in the current directory is read and processed, the function looks for an index file. If an index file is found, the process repeats itself in the directory referred to by the index.

# Transaction Logging API

The following structure controls certain operations of eXtremeDB-TL.

```
typedef struct log_args_ {  
    char * Filepath;  
    uint4 maxlogsize;  
    uint4 maxfiles;  
    uint2 flags;  
    uint2 flush_depth;  
    flush_timer_h flush_timer;  
} log_args_t, *log_args_h;
```

|  |  |
|--|--|
| <code>char * Filepath</code>           | The path to the log “base” directory. The runtime creates a directory hierarchy to increase the performance of disk-based procedures. The root of this hierarchy is set by the application.  |
| <code>uint4 maxlogsize</code>          | The maximum size of each log file. When the max size is reached, the current log file is closed and a new one is created. The actual file size might slightly exceed the maxlogsize value since the file is only closed after a complete transaction is written into it.   |
| <code>uint4 maxfiles</code>            | The maximum number of the log files created in each directory. When this number is reached, the next new log file is created in a new subdirectory of <code>Filepath</code> .  |
| <code>uint2 flags</code>               | Various bit flags (see <code>mcolog.h</code> ) that control the logging process:<br><code>MCO_LOG_CLOSE</code> indicates that the current log file is closed after every write. Please note that the file is closed, not just that the file system buffers are flushed. This has significant performance implications, so don’t set this flag in your production software. Only set this flag for debug purposes.  |
| <code>uint2 flush_depth</code>         | The runtime typically writes data to an internal buffer that the operating system writes to disk on a regular basis. This parameter and the <code>flush_timer</code> parameter control how frequently the runtime clears the buffers for the log file and, consequently, causes all buffered data to be written to disk. This parameter instructs the runtime to flush buffers after the <code>flush_depth</code> number of database commits. The file system may not preserve the content of the file buffers upon system failure. Thus, if <code>flush_depth &gt; 1</code> , there is a chance that up to <code>flush_depth</code> number of transactions will not be actually recorded to the log files. If your application must ensure that every transaction is recorded, set the <code>flush_depth</code> to 1. Setting it to zero means the buffers will be flushed at the file system’s discretion. |
| <code>flush_timer_h flush_timer</code> | This parameter instructs the runtime to flush the log buffers based on a timer. The timer procedure is external to the runtime and should be provided by the   |

|  |  |
|--|--|
|  | application. Refer to the example below. |
|--|--|

## ***mco\_log\_start***

`MCO_RET mco_log_start ( mco_db_h dbh, log_args_h args );`

|                              |  |
|------------------------------|--|
| <code>mco_db_h dbh</code>    | The database handle returned by <code>mco_db_connect()</code> ;  |
| <code>log_args_h args</code> | Pointer to the structure that contains log control parameters. This structure declaration could be found in the <code>mcolig.h</code> file |

The function initializes transaction logging. The database must be created and connected to before the function is called. The logging process is turned on and off (via `mco_log_stop()`) at the application's discretion.

**Note:** Transaction logging is not turned on by default, even if the application is linked with the transaction logging library. You must *explicitly* start transaction logging after the database is open and the application has obtained a connection handle to it. Also note that `mco_log_start()` internally creates a database checkpoint file.

The function returns `MCO_S_OKAY` if the initialization was successful, or one of the error codes described in the `mcolog.h` file.

Example:

```
mco_bool FlushTimer (struct flush_timer_* handle)
{
    if(handle->flush_time)
    {
        if( handle->time_elapsed == 0 )
            handle->time_elapsed =
                mco_system_get_current_time();

        if( (mco_system_get_current_time() -
            handle->time_elapsed) > handle->flush_time)
        {
            handle->time_elapsed = 0;
            return TRUE;
        }
    }
    return FALSE;
}

main()
{
    log_args_t  args;

    args.Filepath    = LOG_FILEPATH;
    args.maxlogsize  = LOG_MAX_FILESIZE;
    args.maxfiles    = LOG_MAX_FILES_IN_DIR;
    args.flags       = 1;
    args.flush_depth = flush_depth;
    args.flush_timer = &flush_timer;
    flush_timer.flush_time = flush_time;
    flush_timer.TimerProc = FlushTimer;

    rc = mco_log_start(db, &args );
}
```

## ***mco\_log\_stop***

**MCO\_RET** `mco_log_stop( mco_db_h dbh );`

|                           |   |
|---------------------------|---|
| <code>mco_db_h dbh</code> | The database handle returned by <code>mco_db_connect()</code> ; |
|---------------------------|---|

This function turns off transaction logging. It returns `MCO_S_OKAY` if successful or an error code described in the `mcolog.h`

**Note:** An application can start and stop transaction logging at any time after the database has been connected to. The following fragment illustrates this:

Example:

```
main()
{
    mco_db_h    db;
    log_args_t  args;

    open_database();
    mco_db_connect(dbname, &db);

    /* this is a time-critical portion of code,
     * which does not require transaction logging
     */
    bootstrap_stb();

    /* fill out the log initialization structure */
    init_args(&args);

    /* main application loop */
    while (1)
    {
        /* turn the transaction logging on */
        mco_log_start(db, &args);

        process_regular_programming();
        if (new_channel_advertised())
        {
            /* a new time-critical portion of code.
             * We should interrupt logging for awhile
             * and process new data purely in memory
             */
            mco_log_stop(db);
            accept_new_channel_programming();

            /* done with the time-critical processing
             * start logging again
             */
            mco_log_start(db, &args);

            /* make sure that the latest changes
             * are "checkpointed"
             */
            mco_log_save(db, mode);
        }
    }
    mco_log_stop(db);
}
```

## ***mco\_log\_recover***

**MCO\_RET** `mco_log_recover` ( `mco_db_h` dbh );

|                           |   |
|---------------------------|---|
| <code>mco_db_h</code> dbh | The database handle returned by <code>mco_db_connect()</code> ; |
|---------------------------|---|

This function performs database recovery from the current transaction log. If the transaction log is not found, the function does nothing and returns.

`mco_log_recover()` can return the following values:

|                                |  |
|--------------------------------|--|
| <b>MCO_S_OKAY</b>              | The recovery was successful;   |
| <b>MCO_E_LOG_IMG_NOT_FOUND</b> | No log is found. In this case the recovery is impossible and the database must be started over.  |
| <b>MCO_E_LOG_BROKEN_RECORD</b> | A corrupted or unfinished log record has been found during the recovery. All log records that precede the corrupted one are applied to the database image. |

Example:

```
main()
{
    mco_db_h    db;
    log_args_t  args;

    open_database(dbname);
    mco_db_connect(dbname, &db);
    init_args(&args);

    /* turn the transaction logging on */
    mco_start_log(db, &args);

    /* attempting to recover the database */
    rc = mco_log_recover(db);

    if(rc == MCO_S_OKAY )
        printf("Successful recovery\n");
    else if(rc == MCO_E_LOG_IMG_NOT_FOUND )
        printf("Log does not exist\n");
    else if(rc == MCO_E_LOG_BROKEN_RECORD)
        printf("Log is incomplete, partial recovery\n");

    process_data();

    mco_log_stop(db);
}
```

## ***mco\_log\_save***

**MCO\_RET** `mco_log_save( mco\_db\_h dbh, mco\_bool delete_old_log );`

|   |  |
|---|--|
| <a href="#">mco_db_h</a> dbh            | The database handle returned by <b>mco_db_connect()</b>  |
| <a href="#">mco_bool</a> delete_old_log | This flag indicates whether to delete old log files synchronously or asynchronously. After the checkpoint, old log files are no longer needed and should be deleted from the disk. This procedure could be done synchronously in the context of the same thread, or asynchronously in the context of a different thread. If the <i>delete_old_log</i> flag is set to TRUE, the old log is synchronously deleted. Please note that this could be a rather lengthy procedure and delay the execution of the program. To avoid this effect, the <i>deferred log cleanup</i> can be used. The application needs to set the <i>delete_old_log</i> flag to FALSE and call the <b>mco_log_deferred_deletion()</b> API in a separate thread. The <code>mco_log_save()</code> function will, in this case, wake up the thread designated to clean up the old log files. |

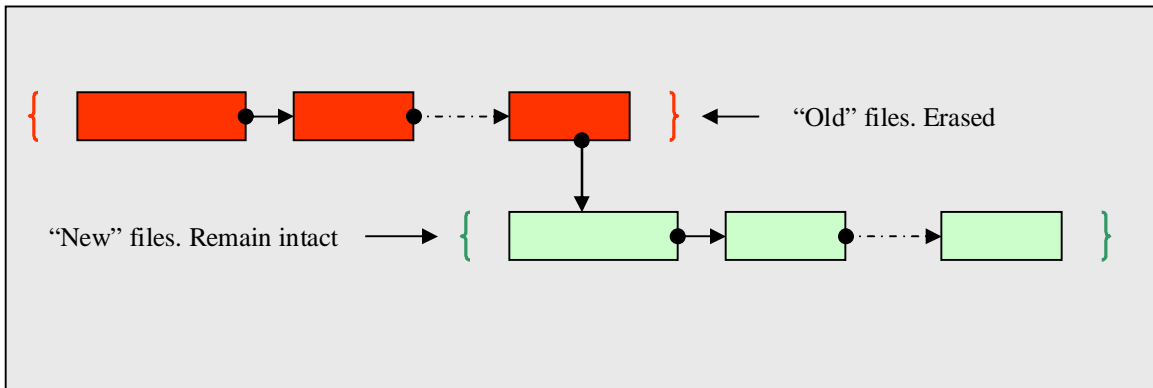
The function returns **MCO\_S\_OKAY** if successful, or an error code otherwise (mcolog.h)

## ***mco\_log\_delete\_old***

```
MCO_RET mco_log_delete_old ( mco_db_h dbh );
```

|                           |   |
|---------------------------|---|
| <code>mco_db_h dbh</code> | The database handle returned by <code>mco_db_connect()</code> |
|---------------------------|---|

This function deletes the old log files. The last checkpoint file created by the `mco_log_save()` and all individual log files created after the last checkpoint are considered “new”; all other checkpoint and log files are considered “old” and are deleted by the `mco_log_delete_old()` API.



## ***mco\_log\_deferred\_deletion***

**MCO\_RET** `mco_log_deferred_deletion( mco_db_h dbh );`

|                           |   |
|---------------------------|---|
| <code>mco_db_h dbh</code> | The database handle returned by the <code>mco_db_connect()</code> |
|---------------------------|---|

This function deletes old log files asynchronously. An application should create a separate “clean-up” thread and call `mco_log_deferred_deletion()` in the context of this thread. `mco_log_deferred_deletion()` sleeps on a semaphore that is created by the eXtremeDB runtime. When `mco_log_save()` signals the semaphore, the clean-up thread is awoken and the old log files are deleted.

Example:

```
/* clean up thread function*/
THREAD_PROC_DEFINE( deferred_deletion, p ) {

    /* blocks here */
    mco_log_deferred_deletion( db );
}

main ()
{
    mco_db_h dbh;
    log_args_t args;
    ...
    /* initialize the database */
    init_database();

    /* initialize transaction logging */
    mco_log_start(dbh, &args);

    /* start the "clean-up" thread */
    THREAD_PROC_START( deferred_deletion, 0, &td );
    ...
}
```

If successful the function executes in a loop and never returns control back to the application. If the initialization was unsuccessful an error code is returned (see `mcollog.h`)

## ***mco\_log\_set\_compression***

```
MCO_RET mco_log_set_compression (  
    mco_db_h dbh,  
    mco_compress compress_func,  
    mco_decompress decompress_func );
```

|   |   |
|---|---|
| <code>mco_db_h dbh</code>                   | The database handle returned by the <b>mco_db_connect()</b> |
| <code>mco_compress compress_func</code>     | Pointer to the user-defined compression routine             |
| <code>mco_decompress decompress_func</code> | Pointer to the user-defined decompression routine           |

This function turns on the transaction log compression mechanism. If compression is on, the transaction data written to the log is compressed. The compression procedure itself must be supplied by the application. The following are the user-defined compression/decompression prototypes:

```
typedef int (*mco_compress)( void * from, uint4 length, void * to);
```

|                           |   |
|---------------------------|---|
| <code>void * from</code>  | This is the pointer to the input buffer |
| <code>uint4 length</code> | The length of the input buffer in bytes |
| <code>void * to</code>    | Pointer to the output buffer            |

The function must return the actual size of the output data in bytes.

```
typedef int (*mco_decompress)( void * from, uint4 length, void * to);
```

|                           |  |
|---------------------------|--|
| <code>void * from</code>  | This is the pointer to the input buffer with compressed data |
| <code>uint4 length</code> | The length of the input buffer in bytes                      |
| <code>void * to</code>    | Pointer to the output buffer that contains uncompressed data |

The function must return the actual size of the output data in bytes.

To turn the compression off, an application should pass zeros as arguments to the compression / decompression routines:

```
mco_log_set_compression (db, 0, 0);
```

# DDL Requirements

The schema for a database that might be use the transaction logging must include the *auto\_oid* declaration.

```
declare auto_oid [ESTIMATED_NBR_OBJECTS];
```

Whether the database actually uses the transaction logging is determined at run-time. When the data definition is compiled for transaction logging, the eXtremeDB DDL compiler inserts an 8-byte unique identifier (called auto\_OID, not to be confused with the field type autoid) into each object.