Disk Based Hashtables (DBH) 64 bit

Name

Disk Based Hashtables (DBH) 64 bit — Library to create and manage hash tables residing on disk. Associations are made between keys and values so that for a given a key the value can be found and loaded into memory quickly. Being disk based allows for large and persistent hashes. 64 bit support allows for hashtables with sizes over 4 Gigabytes on 32 bit systems. Quantified key generation allows for minimum access time on balanced multidimensional trees.

Stability Level

Stable, unless otherwise indicated

Synopsis

```
#include <dbh.h>

#define             DBH_CREATE
#define             DBH_DATA                        (dbh)
#define             DBH_DATA_SPACE                  (dbh)
#define             DBH_ERASED_SPACE                (dbh)
#define             DBH_FILE_VERSION
#define             DBH_FORMAT_SPACE                (dbh)
#define             DBH_KEY                         (dbh)
#define             DBH_KEYLENGTH                   (dbh)
#define             DBH_MAXIMUM_RECORD_SIZE         (dbh)
#define             DBH_PARALLEL_SAFE
#define             DBH_PATH                        (dbh)
#define             DBH_READ_ONLY
#define             DBH_RECORDS                     (dbh)
#define             DBH_RECORD_SIZE                 (dbh)
#define             DBH_THREAD_SAFE
#define             DBH_TOTAL_SPACE                 (dbh)
#define             DBH_VERSION
void                (*DBHashFunc)                   (DBHashTable *dbh);
void                (*DBHashFunc2)                  (DBHashTable *dbh,
                                                     void *data);
struct              DBHashTable;
#define             FILE_POINTER
DBHashTable *       dbh_new                         (const char *path,
```

```
                                                    unsigned char *key_length,
                                                    int flags);
DBHashTable *       dbh_open                       (const char *path);
DBHashTable *       dbh_open_ro                    (const char *path);
DBHashTable *       dbh_create                     (const char *path,
                                                    unsigned char key_length);
int                 dbh_close                      (DBHashTable *dbh);
int                 dbh_destroy                    (DBHashTable *dbh);
int                 dbh_erase                      (DBHashTable *dbh);
int                 dbh_unerase                    (DBHashTable *dbh);
FILE_POINTER        dbh_update                     (DBHashTable *dbh);
FILE_POINTER        dbh_load                       (DBHashTable *dbh);
unsigned char       dbh_load_address               (DBHashTable *dbh,
                                                    FILE_POINTER currentseek);
FILE_POINTER        dbh_load_child                 (DBHashTable *dbh,
                                                    unsigned char key_index);
FILE_POINTER        dbh_load_parent                (DBHashTable *dbh);
void                dbh_set_data                   (DBHashTable *dbh,
                                                    void *data,
                                                    FILE_POINTER size);
void                dbh_set_key                    (DBHashTable *dbh,
                                                    unsigned char *key);
void                dbh_set_recordsize             (DBHashTable *dbh,
                                                    int record_size);
int                 dbh_set_size                   (DBHashTable *dbh,
                                                    FILE_POINTER size);
int                 dbh_settempdir                 (DBHashTable *dbh,
                                                    char *temp_dir);
                    dbh_lock_t;
int                 dbh_clear_locks                (DBHashTable *dbh);
int                 dbh_set_lock_timeout           (int seconds);
int                 dbh_get_lock_timeout           (void);
int                 dbh_set_parallel_lock_attempt_limit (DBHashTable *dbh,
                                                    int limit);
int                 dbh_set_parallel_lock_timeout  (DBHashTable *dbh,
                                                    int seconds);
int                 dbh_lock_read                  (DBHashTable *dbh);
int                 dbh_unlock_read                (DBHashTable *dbh);
int                 dbh_lock_write                 (DBHashTable *dbh);
int                 dbh_unlock_write               (DBHashTable *dbh);
int                 dbh_mutex_lock                 (DBHashTable *dbh);
int                 dbh_mutex_unlock               (DBHashTable *dbh);
FILE_POINTER        dbh_find                       (DBHashTable *dbh,
                                                    int n);
int                 dbh_fanout                     (DBHashTable *dbh,
                                                    DBHashFunc operate,
                                                    unsigned char *key1,
```

```
                                                 unsigned char *key2,
                                                 unsigned char ignore_portion)
int             dbh_sweep                       (DBHashTable *dbh,
                                                 DBHashFunc operate,
                                                 unsigned char *key1,
                                                 unsigned char *key2,
                                                 unsigned char ignore_portion)
int             dbh_foreach                     (DBHashTable *dbh,
                                                 DBHashFunc2 operate,
                                                 void *data);
int             dbh_foreach_fanout              (DBHashTable *dbh,
                                                 DBHashFunc operate);
int             dbh_foreach_sweep               (DBHashTable *dbh,
                                                 DBHashFunc operate);
void            dbh_exit_fanout                 (DBHashTable *dbh);
void            dbh_exit_sweep                  (DBHashTable *dbh);
int             dbh_prune                       (DBHashTable *dbh,
                                                 unsigned char *key,
                                                 unsigned char subtree_length)
int             dbh_unprune                     (DBHashTable *dbh,
                                                 unsigned char *key,
                                                 unsigned char subtree_length)
void            dbh_regen_fanout                (DBHashTable **dbh);
void            dbh_regen_sweep                 (DBHashTable **dbh);
void            dbh_genkey                      (unsigned char *key,
                                                 unsigned char length,
                                                 unsigned int n);
void            dbh_genkey0                     (unsigned char *key,
                                                 unsigned char length,
                                                 unsigned int n);
void            dbh_genkey2                     (unsigned char *key,
                                                 unsigned char length,
                                                 unsigned int n);
void            dbh_orderkey                    (unsigned char *key,
                                                 unsigned char length,
                                                 unsigned int n,
                                                 unsigned char base);
struct          dbh_header_t;
int             dbh_info                        (DBHashTable *dbh);
int             dbh_writeheader                 (DBHashTable *dbh);
```

Description

A DBHashTable provides associations between keys and values which is optimized so that given a key, the associated value can be found very quickly.

Note that only one hash record is loaded from disk to memory at any given moment for a DBHashTable. Both keys and values should be copied into the DBHashTable record, so they need not exist for the lifetime of the DBHashTable. This means that the use of static strings and temporary strings (i.e. those created in buffers and those returned by GTK+ widgets) should be copied with `dbh_set_key()` (see [dbh_set_key []], page 14) and `dbh_set_data()` (see [dbh_set_data []], page 14) into the DBHashTable record before being inserted.

You must be careful to ensure that copied key length matches the defined key length of the DBHashTable, and also that the copied data does not exceed the maximum length of the DBHashTable record (1024 bytes by default, and expandable by `dbh_set_size()` (see [dbh_set_size []], page 15)). If the DBHashTable record length is to be variable, be sure to set the appropriate length before each `dbh_update()` (see [dbh_update []], page 12), with `dbh_set_recordsize()` (see [dbh_set_recordsize []], page 14), otherwise the record length need only be set before the first `dbh_update()` (see [dbh_update []], page 12).

To create a DBHashTable, use `dbh_new()` (see [dbh_new []], page 10).

A DBHashTable may be opened (either new or existing) in read-only mode, parallel-safe mode or thread-safe mode.

To insert a key and value into a DBHashTable, use `dbh_update()` (see [dbh_update []], page 12). The DBHashTable will not be modified until this command is given. All changes to the current DBHashTable record only reside in memory. `dbh_update()` (see [dbh_update []], page 12) is necessary to commit the changes to the DBHashTable.

To lookup a value corresponding to a given key, use `dbh_load()` (see [dbh_load []], page 12).

To erase and unerase a key and value, use `dbh_erase()` (see [dbh_erase []], page 12) and `dbh_unerase()` (see [dbh_unerase []], page 12).

To call a function for each key and value pair (using a sweep route) use `dbh_foreach_sweep()` (see [dbh_foreach_sweep []], page 21) and `dbh_sweep()` (see [dbh_sweep []], page 20).

To call a function for each key and value pair (using a fanout route) use `dbh_foreach_fanout()` (see [dbh_foreach_fanout []], page 21) and `dbh_foreach_fanout()` (see [dbh_foreach_fanout []], page 21).

To destroy a DBHashTable use `dbh_destroy()` (see [dbh_destroy []], page 11).

This is dbh version 2, incompatible with dbh version 1 files. The main difference between the two version is the handling of file pointers. In version 1, file pointers were 32 bits in length, while in version 2, file pointers are 64 bits in length. This allows for DBHashTables with sizes greater than 2 GBytes.

'Quantified numbers' are an alternate way to view the set of 'natural numbers' {1, 2, 3, ...} where order is defined in two levels. In 'natural numbers' there is only one level of order (defined by the `>` boolean operator). In 'quantified numbers' the first level of order is defined by the 'cuanta' or quantity. The 'cuanta' is obtained by adding all the digits of the 'quantified number'. Thus, for example, 10022, 5, 32, and 11111 are all equal at the first level of order since they all add up to 5. The second level or order may be obtained in different manners. In functions `dbh_genkey()` (see [dbh_genkey []], page 24) and `dbh_genkey2()` (see [dbh_genkey2 []], page 24) the corresponding order of the 'natural numbers' from which they are associated is *not conserved*.

In `dbh_orderkey()` (see [dbh_orderkey []], page 25) the corresponding order of the 'natural numbers' from which they are associated *is conserved*, but at a price. The base, or maximum value each digit may reach, must be defined. This effectively puts a limit on the number of keys which may be generated for a given number of digits.

When a `DBHashTable` (see [struct DBHashTable], page 9) is constructed with 'quantified' keys, the maximum amount of disk access instructions generated to access any given record is equal to the 'cuanta' of the quantified number represented by the key. This allows a `DBHashTable` (see [struct DBHashTable], page 9) to be constructed with minimum access time across all records.

Details

DBH_CREATE

```
#define             DBH_CREATE
```

Bit flag for `dbh_new()` (see [dbh_new []], page 10) to create a new dbh file on disk, overwriting any file with the same name and cleansing all locks.

DBH_DATA()

```
#define             DBH_DATA(dbh)
```

This macro returns a pointer to the current `DBHashTable` (see [struct DBHashTable], page 9) data area.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_DATA_SPACE()

```
#define             DBH_DATA_SPACE(dbh)
```

This macro returns the amount of bytes taken up by valid data in the `DBHashTable` (see [struct DBHashTable], page 9).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_ERASED_SPACE()

```
#define             DBH_ERASED_SPACE(dbh)
```

This macro returns the amount of bytes taken up by erased data in the `DBHashTable` (see [struct DBHashTable], page 9).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_FILE_VERSION

```
#define DBH_FILE_VERSION  "DBH_2.0/64bit"
```

Disk Based Hashtables library file version compatibility

DBH_FORMAT_SPACE()

```
#define             DBH_FORMAT_SPACE(dbh)
```

This macro returns the total amount of bytes taken up by the format of the `DBHashTable` (see [struct DBHashTable], page 9).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_KEY()

```
#define             DBH_KEY(dbh)
```

This macro returns a pointer to the current `DBHashTable` (see [struct DBHashTable], page 9) key area.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_KEYLENGTH()

```
#define             DBH_KEYLENGTH(dbh)
```

This macro returns the keylenth in bytes associated to the `DBHashTable` (see [struct DBHashTable], page 9). The value is fixed when the `DBHashTable` (see [struct DBHashTable], page 9) is created with `dbh_new` (see [dbh_new []], page 10).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_MAXIMUM_RECORD_SIZE()

```
#define             DBH_MAXIMUM_RECORD_SIZE(dbh)
```

This macro returns the maximum allocated space for data in the current `DBHashTable` (see [struct DBHashTable], page 9) record.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_PARALLEL_SAFE

```
#define             DBH_PARALLEL_SAFE
```

Bit flag for `dbh_new()` (see [dbh_new []], page 10) to use if more than one heavy weight process will be accessing the same `DBHashTable` (see [struct DBHashTable], page 9) in write mode. If no process will be writing to the `DBHashTable` (see [struct DBHashTable], page 9), then `DBH_READ_ONLY` (see [DBH_READ_ONLY], page 7) is enough and faster since each process will hold a separate memory allocation for the `DBHashTable` (see [struct DBHashTable], page 9) pointer.

DBH_PATH()

```
#define             DBH_PATH(dbh)
```

This macro returns a pointer to a string containing the path to the current `DBHashTable` (see [struct DBHashTable], page 9).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_READ_ONLY

```
#define             DBH_READ_ONLY
```

Bit flag for `dbh_new()` (see [dbh_new []], page 10) to open an existing dbh file on disk in read only mode.

DBH_RECORDS()

```
#define             DBH_RECORDS(dbh)
```

This macro returns the number of records in the `DBHashTable` (see [struct DBHashTable], page 9).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_RECORD_SIZE()

```
#define             DBH_RECORD_SIZE(dbh)
```

This macro returns the size of the current record loaded in memory. If no record has been loaded, then the return value is not defined.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_THREAD_SAFE

```
#define             DBH_THREAD_SAFE
```

Bit flag for `dbh_new()` (see [dbh_new []], page 10) to use if more than one thread will be accessing the same `DBHashTable` (see [struct DBHashTable], page 9) in write mode in parallel. DBH function calls which may be racing each other in different threads should be enclosed within a `dbh_mutex_lock()` (see [dbh_mutex_lock []], page 18) and `dbh_mutex_unlock()` (see [dbh_mutex_unlock []], page 19). Each DBH table opened with the `DBH_THREAD_SAFE` (see [DBH_THREAD_SAFE], page 8) attribute will have a specific mutex for this function. If threads are to access the same `DBHashTable` (see [struct DBHashTable], page 9) in read mode only, then `DBH_READ_ONLY` (see [DBH_READ_ONLY], page 7) and separate memory allocation for each thread's `DBHashTable` (see [struct DBHashTable], page 9) pointer is more than enough and faster.

When `DBH_THREAD_SAFE` (see [DBH_THREAD_SAFE], page 8) is specified, `dbh_new()` (see [dbh_new []], page 10) is automatically mutex locked until function completes. The function `dbh_close()` (see [dbh_close []], page 11) is also automatically locked until completion on tables opened with the `DBH_THREAD_SAFE` (see [DBH_THREAD_SAFE], page 8) attribute.

DBH_TOTAL_SPACE()

```
#define             DBH_TOTAL_SPACE(dbh)
```

This macro returns the total amount of bytes taken up by the `DBHashTable` (see [struct DBHashTable], page 9).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBH_VERSION

```
#define DBH_VERSION  "5.0.15"
```

Disk Based Hashtables library version

DBHashFunc ()

```
void (*DBHashFunc) (DBHashTable *dbh);
```

Pointer to function to apply during `dbh_sweep()` (see [dbh_sweep []], page 20), `dbh_fanout()` (see [dbh_fanout []], page 19), `dbh_foreach_sweep()` (see [dbh_foreach_sweep []], page 21) and `dbh_foreach_fanout()` (see [dbh_foreach_fanout []], page 21).

This function will be applied to all data records involved in the sweep or fanout process

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *)

DBHashFunc2 ()

```
void (*DBHashFunc2) (DBHashTable *dbh,
 void *data);
```

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *) Pointer to function to apply during `dbh_sweep()` (see [dbh_sweep []], page 20), `dbh_fanout()` (see [dbh_fanout []], page 19), `dbh_foreach_sweep()` (see [dbh_foreach_sweep []], page 21) and `dbh_foreach_fanout()` (see [dbh_foreach_fanout []], page 21).

`data` : pointer to other data to be passed to function

This function will be applied to all data records involved in the sweep or fanout process

struct DBHashTable

```
struct DBHashTable {
  unsigned char branches;
  FILE_POINTER bytes_userdata;
  unsigned char *key;
  void *data;
  int fd;
  dbh_header_t *head_info;
  char *path;
};
```

`DBHashTable` (see [struct DBHashTable], page 9) is a data structure containing the record information for an open `DBHashTable` (see [struct DBHashTable], page 9) file.

unsigned [Cross reference to non-existant ID “char”] `branches`;
: Maximum toplevel branches

`FILE_POINTER` (see [FILE_POINTER], page 10) `bytes_userdata`;
: size of data record

unsigned [Cross reference to non-existant ID “char”] *`key`;
: access key

[Cross reference to non-existant ID “void”] *`data`;
: record data pointer

[Cross reference to non-existant ID “int”] `fd`;
: file descriptor

`dbh_header_t` (see [struct dbh_header_t], page 25) *`head_info`;
: nonvolatile header information

[Cross reference to non-existant ID “char”] *`path`;
: file path

FILE_POINTER

```
#define             FILE_POINTER
```

Architecture independent 64 bit integer type

dbh_new ()

```
DBHashTable *       dbh_new                         (const char *path,
                                                     unsigned char *key_length,
                                                     int flags);
```

Open or create an existing DBH table. Flag is bitwise or of the following: `DBH_CREATE` (see [DBH_CREATE], page 5), `DBH_READ_ONLY` (see [DBH_READ_ONLY], page 7), `DBH_THREAD_SAFE` (see [DBH_THREAD_SAFE], page 8), `DBH_PARALLEL_SAFE` (see [DBH_PARALLEL_SAFE], page 7). (since 4.7.6)

`path` :
: Path on disk where DBHashTable resides.

`key_length` :
: A pointer to store the length of the key to access the DBHashTable.

`flags` :
: Bitwise or of DBH_CREATE, DBH_READ_ONLY, DBH_THREAD_SAFE, DBH_PARALLEL_SAFE

Returns :
: A pointer to the newly opened DBHashTable, or NULL if it fails.

dbh_open ()

```
DBHashTable *       dbh_open                        (const char *path);
```

Warning

‘`dbh_open`’ is deprecated and should not be used in newly-written code. Use `dbh_new()` (see [dbh_new []], page 10) instead

Open an existing hash in read-write mode.

`path` :
: Path on disk where DBHashTable resides.

Returns : A pointer to the newly opened `DBHashTable` (see [struct DBHashTable], page 9), or NULL if it fails.

dbh_open_ro ()

```
DBHashTable *       dbh_open_ro                     (const char *path);
```

Warning

'`dbh_open_ro`' is deprecated and should not be used in newly-written code. Use `dbh_new()` (see [dbh_new []], page 10) instead

Open an existing hash in read-only mode.

`path` : Path on disk where DBHashTable resides.

Returns : A pointer to the newly opened read-only DBHashTable, or NULL if it fails.

dbh_create ()

```
DBHashTable *       dbh_create                      (const char *path,
                                                     unsigned char key_length);
```

Warning

'`dbh_create`' is deprecated and should not be used in newly-written code. Use `dbh_new()` (see [dbh_new []], page 10) instead

Create a new hash file (overwriting old version). Creates and opens for writing a new `DBHashTable` (see [struct DBHashTable], page 9). This function will overwrite any file with the specified path, including any previous DBH file. The `key_length` is fixed. If you want variable length, use a g_hash_table to associate quantified keys generated by [Cross reference to non-existant ID "genkey"], and create an extra DBHashTable to save the g_hash. Quantified keys assure that large DBHashes are spread out optimally.

`path` : Path on disk where DBHashTable will reside.

`key_length` :
The length of the key to access the DBHashTable.

Returns : A pointer to the newly created and opened `DBHashTable` (see [struct DBHashTable], page 9), or NULL if it fails.

dbh_close ()

```
int                 dbh_close                       (DBHashTable *dbh);
```

Close hash file (thus flushing io buffer).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 on error, 1 otherwise.

dbh_destroy ()

```
int                 dbh_destroy                         (DBHashTable *dbh);
```

Close an open DBHashTable and erase file from disk. Convenience function that does a close and rm.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_erase ()

```
int                 dbh_erase                           (DBHashTable *dbh);
```

Mark the record currently loaded into memory as erased. If no record is currently loaded, behaviour is undefined.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 on error, 1 otherwise.

dbh_unerase ()

```
int                 dbh_unerase                         (DBHashTable *dbh);
```

This is the opposite of `dbh_erase()` (see [dbh_erase []], page 12). Mark the record currently loaded into memory as unerased. If no record is currently loaded, behaviour is undefined.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 on error, 1 otherwise.

dbh_update ()

```
FILE_POINTER        dbh_update                          (DBHashTable *dbh);
```

Update the current record in memory to the disk based hash. Update function will update erased records as well as unerased records, but if an erased record is updated, it is automatically unerased.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 on error, byte offset of loaded record otherwise.

dbh_load ()

```
FILE_POINTER        dbh_load                                (DBHashTable *dbh);
```

Load a record using the currently set key. This function will also load erased values, except that it will return 0.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 on error, byte offset of loaded record otherwise.

dbh_load_address ()

```
unsigned char       dbh_load_address                        (DBHashTable *dbh,
                                                             FILE_POINTER currentseek);
```

Load a record from hash table directly from byte offset `currentseek`

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`currentseek` :
A byte offset.

Returns : 0 on error, number of branches otherwise.

dbh_load_child ()

```
FILE_POINTER        dbh_load_child                          (DBHashTable *dbh,
                                                             unsigned char key_index);
```

Load the first child of the currently loaded record, on branch identified by `key_index`. Since the number of childs (or branches) of each record is variable, this may be tricky. Top level records have `DBH_KEYLENGTH` (see [DBH_KEYLENGTH[]], page 6) branches. Lower level records have less. Each byte of a key represents a branch on top level records.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`key_index` :
branch number on which to return the child.

Returns : 0 on error, byte offset of loaded record otherwise.

dbh_load_parent ()

```
FILE_POINTER        dbh_load_parent                    (DBHashTable *dbh);
```

Load the parent of the currently loaded record.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 on error, byte offset of loaded record otherwise.

dbh_set_data ()

```
void                dbh_set_data                       (DBHashTable *dbh,
                                                        void *data,
                                                        FILE_POINTER size);
```

This function copies the user data into the current `DBHashTable` (see [struct DBHashTable], page 9) record and along with function `dbh_set_key()` (see [dbh_set_key []], page 14), makes the current `DBHashTable` (see [struct DBHashTable], page 9) record ready for the `dbh_update()` (see [dbh_update []], page 12) function to commit to the actual `DBHashTable` (see [struct DBHashTable], page 9) on disk.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`data` : Pointer to the data to copy to the current `DBHashTable` (see [struct DBHashTable], page 9) record

`size` : The amount of bytes to copy to the current `DBHashTable` (see [struct DBHashTable], page 9) record

dbh_set_key ()

```
void                dbh_set_key                        (DBHashTable *dbh,
                                                        unsigned char *key);
```

This function sets the key of the current DBHashTable record.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`key` : The key to set as the current DBHashTable record key.

dbh_set_recordsize ()

```
void                dbh_set_recordsize                 (DBHashTable *dbh,
                                                        int record_size);
```

This sets the recordsize of the the data in the current `DBHashTable` (see [struct DBHashTable], page 9) record. It is called implicitly by calling `dbh_set_data()` (see

[dbh_set_data []], page 14). It is very important to call this function. Unpredictable results will follow if record_size is not set. `DBHashTable` (see [struct DBHashTable], page 9) records are variable in length, so use this function at least once if you are planning to use fixed length records. This function is not needed if `dbh_set_data()` (see [dbh_set_data []], page 14) is used to set the record data.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`record_size` :
The amount of bytes in the current `DBHashTable` (see [struct DBHashTable], page 9) record.

dbh_set_size ()

```
int                 dbh_set_size                    (DBHashTable *dbh,
                                                     FILE_POINTER size);
```

Defines the maximum amount of memory to be allocated to the `DBHashTable` (see [struct DBHashTable], page 9) records. This is nonvolatile information which need to be set only once. The default is 1Kbyte.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`size` : size in bytes.

Returns : 0 on error, 1 otherwise.

dbh_settempdir ()

```
int                 dbh_settempdir                  (DBHashTable *dbh,
                                                     char *temp_dir);
```

Sets the temporary directory to be used by `dbh_regen_sweep()` (see [dbh_regen_sweep []], page 23) or `dbh_regen_fanout()` (see [dbh_regen_fanout []], page 23). It is usually best to set temporary directory on the same filesystem device. The default value for the temporary directory is the directory where `dbh` is located. To reset to default value, send NULL as the `temp_dir`

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`temp_dir` :
path to temporary directory to use.

Returns : 0 if error, 1 otherwise

dbh_lock_t

```
typedef struct {
    pid_t write_lock;
    int write_lock_count;
    int read_lock_count;
} dbh_lock_t;
```

[Cross reference to non-existant ID "pid-t"] `write_lock`;
: PID of process holding the write lock or zero.

[Cross reference to non-existant ID "int"] `write_lock_count`;
: Number of write locks the PID hold (write locks are recursive).

[Cross reference to non-existant ID "int"] `read_lock_count`;
: Number of read locks on DBH table.

dbh_clear_locks ()

```
int                 dbh_clear_locks                 (DBHashTable *dbh);
```

`Returns`: 0 if error, 1 otherwise

Clear dbh file locks associated to `DBHashTable` (see [struct DBHashTable], page 9) Use this function to clean up persistent file locks

(since 4.7.6)

`dbh` :
: A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

dbh_set_lock_timeout ()

```
int                 dbh_set_lock_timeout            (int seconds);
```

Sets the default time for obtaining a read/write lock in parallel safe mode. The default value is zero, which means there is no timeout. If there is no timeout, file locking will block until lock is secured. Locks may persist beyond program life and may be stale if program crashed before unlocking was performed. Does not affect currently open dbh files. If the value for a currently open dbh file is to be modified, use `dbh_set_parallel_lock_timeout()` (see [dbh_set_parallel_lock_timeout []], page 17) as well.

`seconds` :
: Timeout default for obtaining a read/write lock in parallel safe mode.

Returns :
: 0 if error, 1 otherwise

dbh_get_lock_timeout ()

```
int                 dbh_get_lock_timeout                (void);
```

Gets the default time for obtaining a read/write lock in parallel safe mode. The default value is zero, which means there is no timeout. If there is no timeout, file locking will block until lock is secured. Locks may persist beyond program life and may be stale if program crashed before unlocking was performed.

Returns : the default timeout in seconds to secure a read/write lock in parallel safe mode.

dbh_set_parallel_lock_attempt_limit ()

```
int                 dbh_set_parallel_lock_attempt_limit (DBHashTable *dbh,
                                                         int limit);
```

Warning

'`dbh_set_parallel_lock_attempt_limit`' is deprecated and should not be used in newly-written code. Use `dbh_set_parallel_lock_timeout()` (see [dbh_set_parallel_lock_timeout []], page 17) instead. As of 5.0.10, this function is inoperative.

Sets the limit on the attempts to lock a parallel protected dbh file lock before considering the lock to be stale. Stale locks may occur when the calling program crashes while the lock is set in either read or write mode. Lock will persist in shared memory beyond program crash. Lock may be removed manually, or a lock attempt limit on the number of tries specified to remove the lock automatically. Each lock attempt limit is equal to 1/10th of a second (1E+08 nanoseconds). If limit is set to zero, then lock attempts will continue indefinitely.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`limit` : Number of attempts to lock a parallel protected file lock before removing lock.

Returns : 0 if error, 1 otherwise

dbh_set_parallel_lock_timeout ()

```
int                 dbh_set_parallel_lock_timeout       (DBHashTable *dbh,
                                                         int seconds);
```

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`seconds` : Number of second to try to lock a parallel protected file before failing. A value of zero means function will block until lock is obtained.

Returns : 0 if error, 1 otherwise

dbh_lock_read ()

```
int                 dbh_lock_read                       (DBHashTable *dbh);
```

Attempts to get a read lock on the dbh file. A file may have any number of readlocks as long as no write lock is set. If `dbh_set_parallel_lock_timeout()` (see [dbh_set_parallel_lock_timeout []], page 17) is set to zero (that's the default) this function will block until lock is secured.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_unlock_read ()

```
int                 dbh_unlock_read                     (DBHashTable *dbh);
```

Releases a read lock on the dbh file.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_lock_write ()

```
int                 dbh_lock_write                      (DBHashTable *dbh);
```

Attempts to get a write lock on the dbh file. A file can only have one write lock, and when write lock is set, no read locks may be secured. If `dbh_set_parallel_lock_timeout()` (see [dbh_set_parallel_lock_timeout []], page 17) is set to zero (that's the default) this function will block until lock is secured.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_unlock_write ()

```
int                 dbh_unlock_write                    (DBHashTable *dbh);
```

Releases a write lock on the dbh file.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_mutex_lock ()

```
int                 dbh_mutex_lock                          (DBHashTable *dbh);
```

Lock the DBHashTable mutex. This is only valid if table was opened with the DBH_THREAD_SAFE flag, Otherwise the function does nothing.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_mutex_unlock ()

```
int                 dbh_mutex_unlock                        (DBHashTable *dbh);
```

Unlock the DBHashTable mutex. This is only valid if table was opened with the DBH_THREAD_SAFE flag, Otherwise the function does nothing.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

dbh_find ()

```
FILE_POINTER        dbh_find                                (DBHashTable *dbh,
                                                             int n);
```

Find the top level subtree FILE_POINTER for the currently loaded record, but ignoring the last `n` branches.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`n` : Number of branches to ignore on top record.

Returns : 0 on error, byte offset of loaded record otherwise.

dbh_fanout ()

```
int                 dbh_fanout                              (DBHashTable *dbh,
                                                             DBHashFunc operate,
                                                             unsigned char *key1,
                                                             unsigned char *key2,
                                                             unsigned char ignore_portion)
```

Apply a function to subtree members of the hash, following a fanout trajectory (horizontally through records).

In order for `dbh_fanout()` (see [dbh_fanout []], page 19) to be extremely fast, you should prepare the `DBHashTable` (see [struct DBHashTable], page 9) for the trajectory with

`dbh_regen_fanout()` (see [dbh_regen_fanout []], page 23) first. This allows for extremely efficient use of hardware and operating system caches.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`operate` : The function to apply to each selected member of the `DBHashTable` (see [struct DBHashTable], page 9)

`key1` : The key from which to start the fanout or NULL if you don't care. Make sure it is a top level node of a subtree with `dbh_find()` (see [dbh_find []], page 19) first.

`key2` : The key which will trigger an exit condition from the sweep, or NULL if don't care.

`ignore_portion` :
The ignored trailing bytes of key1 which will define the magnitud of the subtree to be sweeped, or zero if don't care.

Returns : 0 on error, 1 otherwise.

dbh_sweep ()

```
int                 dbh_sweep                       (DBHashTable *dbh,
                                                     DBHashFunc operate,
                                                     unsigned char *key1,
                                                     unsigned char *key2,
                                                     unsigned char ignore_portion)
```

Apply a function to subtree members of the hash, following a sweep trajectory (vertically through branches).

In order for `dbh_sweep()` (see [dbh_sweep []], page 20) to be extremely fast, you should prepare the `DBHashTable` (see [struct DBHashTable], page 9) for the trajectory with `dbh_regen_sweep()` (see [dbh_regen_sweep []], page 23) first. This allows for extremely efficient use of hardware and operating system caches.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`operate` : The function to apply to each selected member of the `DBHashTable` (see [struct DBHashTable], page 9)

`key1` : The key from which to start the sweep or NULL if you don't care. Make sure it is a top level node of a subtree with `dbh_find()` (see [dbh_find []], page 19) first.

`key2` : The key which will trigger an exit condition from the sweep, or NULL if don't care.

`ignore_portion` :
The ignored trailing bytes of key1 which will define the magnitud of the subtree to be sweeped, or zero if don't care.

Returns : 0 on error, 1 otherwise.

dbh_foreach ()

```
int                 dbh_foreach                         (DBHashTable *dbh,
                                                         DBHashFunc2 operate,
                                                         void *data);
```

Apply a function to each member of the hash, following a sweep trajectory. Sweep is done by traversing the `DBHashTable` (see [struct DBHashTable], page 9) in a vertical direction through all branches.

In order for `dbh_foreach_sweep()` (see [dbh_foreach_sweep []], page 21) to be extremely fast, you should prepare the `DBHashTable` (see [struct DBHashTable], page 9) for the trajectory with `dbh_regen_sweep()` (see [dbh_regen_sweep []], page 23) first. This allows for extremely efficient use of hardware and operating system caches.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`operate` : A `DBHashFunc2()` (see [DBHashFunc2 []], page 9) to execute on all records

`data` : pointer to data passed to `DBHashFunc2()` (see [DBHashFunc2 []], page 9)

Returns : 0 on error, 1 otherwise.

dbh_foreach_fanout ()

```
int                 dbh_foreach_fanout                  (DBHashTable *dbh,
                                                         DBHashFunc operate);
```

Apply a function to each member of the hash, following a fanout trajectory (horizontally through records). `dbh_foreach_fanout()` (see [dbh_foreach_fanout []], page 21) is done by traversing the `DBHashTable` (see [struct DBHashTable], page 9) in a horizontal direction through all records.

In order for `dbh_foreach_fanout()` (see [dbh_foreach_fanout []], page 21) to be extremely fast, you should prepare the `DBHashTable` (see [struct DBHashTable], page 9) for the trajectory with `dbh_regen_fanout()` (see [dbh_regen_fanout []], page 23) first. This allows for extremely efficient use of hardware and operating system caches.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`operate` : A `DBHashFunc()` (see [DBHashFunc []], page 8) to execute on all records

Returns : 0 on error, 1 otherwise.

dbh_foreach_sweep ()

```
int                 dbh_foreach_sweep               (DBHashTable *dbh,
                                                     DBHashFunc operate);
```

Apply a function to each member of the hash, following a sweep trajectory. Sweep is done by traversing the `DBHashTable` (see [struct DBHashTable], page 9) in a vertical direction through all branches.

In order for `dbh_foreach_sweep()` (see [dbh_foreach_sweep []], page 21) to be extremely fast, you should prepare the `DBHashTable` (see [struct DBHashTable], page 9) for the trajectory with `dbh_regen_sweep()` (see [dbh_regen_sweep []], page 23) first. This allows for extremely efficient use of hardware and operating system caches.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`operate` : A `DBHashFunc()` (see [DBHashFunc []], page 8) to execute on all records

Returns : 0 on error, 1 otherwise.

dbh_exit_fanout ()

```
void                dbh_exit_fanout                 (DBHashTable *dbh);
```

Calling this function from within a `DBHashFunc` (see [DBHashFunc []], page 8) will cause an exit of a currently running fanout.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

dbh_exit_sweep ()

```
void                dbh_exit_sweep                  (DBHashTable *dbh);
```

Calling this function from within a `DBHashFunc` (see [DBHashFunc []], page 8) will cause an exit of a currently running sweep.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

dbh_prune ()

```
int                 dbh_prune                       (DBHashTable *dbh,
                                                     unsigned char *key,
                                                     unsigned char subtree_length)
```

Erases a whole subtree from the record currently loaded into memory. Records are not really removed fisically, but rather marked erased so they may be recovered (if not overwritten later on). Records are permanently removed after `DBHashTable` (see [struct DBHashTable], page 9) is reconstructed with `dbh_regen_sweep()` (see [dbh_regen_sweep []], page 23) or `dbh_regen_fanout()` (see [dbh_regen_fanout []], page 23).

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`key` : key of top level record of subtree to erase.

`subtree_length` :
number of branches to erase.

Returns : 0 on error, 1 otherwise.

dbh_unprune ()

```
int                 dbh_unprune                 (DBHashTable *dbh,
                                                 unsigned char *key,
                                                 unsigned char subtree_length)
```

Does the opposite of `dbh_prune()` (see [dbh_prune []], page 22), marking entire subtree as unerased. May fail to work if records have been overwritten since the `dbh_prune()` (see [dbh_prune []], page 22) instruction was issued.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

`key` : key of top level record of subtree to erase.

`subtree_length` :
number of branches to erase.

Returns : 0 on error, 1 otherwise.

dbh_regen_fanout ()

```
void                dbh_regen_fanout            (DBHashTable **dbh);
```

Regenerate the `DBHashTable` (see [struct DBHashTable], page 9), eliminating erased records and optimizing disk access and speed for fanout access. This is done by creating a new `DBHashTable` (see [struct DBHashTable], page 9) where the physical structure matches the logical fanout structure. The temporary directory where the new `DBHashTable` (see [struct DBHashTable], page 9) is created may be set with `dbh_settempdir()` (see [dbh_settempdir []], page 15). Current `DBHashTable` (see [struct DBHashTable], page 9) is closed before removed. New `DBHashTable` (see [struct DBHashTable], page 9) is opened after renamed.

`dbh` : A pointer to a `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : void.

dbh_regen_sweep ()

```
void                dbh_regen_sweep                 (DBHashTable **dbh);
```

Regenerate the `DBHashTable` (see [struct DBHashTable], page 9), eliminating erased records and optimizing disk access and speed for sweep access. This is done by creating a new `DBHashTable` (see [struct DBHashTable], page 9) where the physical structure matches the logical sweep structure. The temporary directory where the new `DBHashTable` (see [struct DBHashTable], page 9) is created may be set with `dbh_settempdir()` (see [dbh_settempdir []], page 15). Current `DBHashTable` (see [struct DBHashTable], page 9) is closed before removed. New `DBHashTable` (see [struct DBHashTable], page 9) is opened after renamed.

`dbh` : A pointer to a `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : void.

dbh_genkey ()

```
void                dbh_genkey                      (unsigned char *key,
                                                     unsigned char length,
                                                     unsigned int n);
```

Obtain a key from a secuential series of natural numbers (positive integers without zero) which does not conserve the order of the natural numbers, but which are optimized for construction of a balanced hash tree. These keys are expressed in quantified numbers. Digits are offset to the `0` symbol (+48).

`key` : The address where to put the generated key

`length` : The key length

`n` : The natural number from which to generate the key

dbh_genkey0 ()

```
void                dbh_genkey0                     (unsigned char *key,
                                                     unsigned char length,
                                                     unsigned int n);
```

Obtain a key from a secuential series of natural numbers (positive integers without zero) which does not conserve the order of the natural numbers, but which are optimized for construction of a balanced hash tree. These keys are expressed in quantified numbers. Digits are not offset.

`key` : The address where to put the generated key

`length` : The key length

`n` : The natural number from which to generate the key

dbh_genkey2 ()

```
void                dbh_genkey2                     (unsigned char *key,
                                                     unsigned char length,
                                                     unsigned int n);
```

Obtain a key from a secuential series of natural numbers (positive integers without zero) which does not conserve the order of the natural numbers, but which are optimized for construction of a balanced hash tree. These keys are expressed in quantified numbers. Digits are offset to the `A` symbol (+65).

`key` : The address where to put the generated key

`length` : The key length

`n` : The natural number from which to generate the key

dbh_orderkey ()

```
void                dbh_orderkey                    (unsigned char *key,
                                                     unsigned char length,
                                                     unsigned int n,
                                                     unsigned char base);
```

Obtain a key from a secuential series of natural numbers (positive integers without zero) which conserves the order of the natural numbers. This function generates a key that belongs to a finite subset of the quantified numbers, but which preserves the order of the natural numbers (up to the supreme, of course)

`key` : The address where to put the generated key

`length` : The key length

`n` : The natural number for which to generate the key

`base` : The number system base to use. This will equal the maximum number of nodes per branch. This —along with the keylength— will also define a maximum number of records for the DBHashTable

struct dbh_header_t

```
struct dbh_header_t {
  unsigned char n_limit;
  unsigned char user_chars[5];
  FILE_POINTER bof;
  FILE_POINTER erased_space;
  FILE_POINTER data_space;
  FILE_POINTER total_space;

  FILE_POINTER records;
```

```
  FILE_POINTER record_length;
  FILE_POINTER user_filepointer[6];
  char version[16];
  char copyright[128];
};
```

`dbh_header_t` (see [struct dbh_header_t], page 25) is the structural information written at the first 256 bytes of a `DBHashTable` (see [struct DBHashTable], page 9) file.

unsigned [Cross reference to non-existant ID "char"] `n_limit`;
: Maximum toplevel branches

unsigned [Cross reference to non-existant ID "char"] `user_chars`[5];
: Five unsigned chars available to user

`FILE_POINTER` (see [FILE_POINTER], page 10) `bof`;
: File pointer to root of tree

`FILE_POINTER` (see [FILE_POINTER], page 10) `erased_space`;
: Amount of bytes marked as erased

`FILE_POINTER` (see [FILE_POINTER], page 10) `data_space`;
: Amount of bytes ocuppied by data

`FILE_POINTER` (see [FILE_POINTER], page 10) `total_space`;
: Amount of bytes ocuppied by data and format

`FILE_POINTER` (see [FILE_POINTER], page 10) `records`;
: Number of records

`FILE_POINTER` (see [FILE_POINTER], page 10) `record_length`;
: Maximum record length

`FILE_POINTER` (see [FILE_POINTER], page 10) `user_filepointer`[6];
: Six 64-bit filepointers available to user

[Cross reference to non-existant ID "char"] `version`[16];
: DBHashTable version compatibility information

[Cross reference to non-existant ID "char"] `copyright`[128];
: DBH sourcecode distribution copyright and download information

dbh_info ()

```
int                 dbh_info                        (DBHashTable *dbh);
```

Prints header information to stdout.

`dbh` :
: A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns :
: 0 if error, 1 otherwise

dbh_writeheader ()

```
int                 dbh_writeheader                 (DBHashTable *dbh);
```

Write out the DBHashTable header information. It is advisable to call this function inmediately after creation of a new DBHashTable to force a buffer flush.

`dbh` : A `DBHashTable` (see [struct DBHashTable], page 9) pointer (`DBHashTable` (see [struct DBHashTable], page 9) *).

Returns : 0 if error, 1 otherwise

See Also

[Cross reference to non-existant ID “GHashTables”]