NAME

db_open − database access methods

SYNOPSIS

```
#include <db.h>

int
db_open(const char *file, DBTYPE type,
        int flags, int mode, DB_ENV *dbenv, DB_INFO *dbinfo, DB **dbpp);

int
DB->close(DB *db, int flags);

int
DB->cursor(DB *db, DB_TXN *txnid, DBC **cursorp);

int
DB->del(DB *db, DB_TXN *txnid, DBT *key, int flags);

int
DB->fd(DB *db, int *fdp);

int
DB->get(DB *db, DB_TXN *txnid, DBT *key, DBT *data, int flags);

int
DB->put(DB *db, DB_TXN *txnid, DBT *key, DBT *data, int flags);

int
DB->sync(DB *db, int flags);

int
DB->stat(DB *db, void *sp, void *(*db_malloc)(size_t), int flags);
```

DESCRIPTION

The DB library is a family of groups of functions that provides a modular programming interface to transactions and record-oriented file access. The library includes support for transactions, locking, logging and file page caching, as well as various indexed access methods. Many of the functional groups (e.g., the file page caching functions) are useful independent of the other DB functions, although some functional groups are explicitly based on other functional groups (e.g., transactions and logging). For a general description of the DB package, see *db_intro*(3).

This manual page describes the overall structure of the DB library access methods.

The currently supported file formats are btree, hashed and recno. The btree format is a representation of a sorted, balanced tree structure. The hashed format is an extensible, dynamic hashing scheme. The recno format supports fixed or variable length records (optionally retrieved from a flat text file).

Storage and retrieval for the DB access methods are based on key/data pairs, or DBT structures as they are typedef'd in the <db.h> include file. See *db_dbt*(3) for specific information on the structure and capabilities of a DBT.

The *db_open* function opens the database represented by *file* for both reading and writing. Files never intended to be shared or preserved on disk may be created by setting the file parameter to NULL.

The *db_open* function copies a pointer to a DB structure (as typedef'd in the <db.h> include file), into the memory location referenced by *dbpp*. This structure includes a set of functions to perform various database actions, as described below. The *db_open* function returns the value of *errno* on failure and 0 on success.

Note, while most of the access methods use *file* as the name of an underlying file on disk, this is not guaranteed. Also, calling *db_open* is a reasonably expensive operation. (This is based on a model where the DBMS keeps a set of files open for a long time rather than opening and closing them on each query.)

The *type* argument is of type DBTYPE (as defined in the <db.h> include file) and must be set to one of

DB_BTREE, DB_HASH, DB_RECNO or DB_UNKNOWN. If *type* is DB_UNKNOWN, the database must already exist and *db_open* will then determine if it is of type DB_BTREE, DB_HASH or DB_RECNO.

The *flags* and *mode* arguments specify how files will be opened and/or created when they don't already exist. The flags value is specified by **or**'ing together one or more of the following values:

Create any underlying files, as necessary. If the files do not already exist and the DB_CREATE flag is not specified, the call will fail.

DB_NOMMAP
: Do not map this file (see *db_mpool*(3) for further information).

DB_RDONLY
: Open the database for reading only. Any attempt to write the database using the access methods will fail regardless of the actual permissions of any underlying files.

Cause the DB handle returned by the *db_open* function to be useable by multiple threads within a single address space, i.e., to be "free-threaded".

DB_TRUNCATE
: "Truncate" the database if it exists, i.e., behave as if the database were just created, discarding any previous contents.

All files created by the access methods are created with mode *mode* (as described in *chmod*(2)) and modified by the process' umask value at the time of creation (see *umask*(2)). The group ownership of created files is based on the system and directory defaults, and is not further specified by DB.

DB_ENV

The access methods make calls to the other subsystems in the DB library based on the *dbenv* argument to *db_open*, which is a pointer to a structure of type DB_ENV (typedef'd in <db.h>). It is expected that applications will use a single DB_ENV structure as the argument to all of the subsystems in the DB package. In order to ensure compatibility with future releases of DB, all fields of the DB_ENV structure that are not explicitly set should be initialized to 0 before the first time the structure is used. Do this by declaring the structure external or static, or by calling the C library routine *bzero*(3) or *memset*(3).

The fields of the DB_ENV structure used by *db_open* are described below. If *dbenv* is NULL or any of its fields are set to 0, defaults appropriate for the system are used where possible.

The following fields in the DB_ENV structure may be initialized before calling *db_open*:

DB_LOG *lg_info;
: If modifications to the file being opened should be logged, the *lg_info* field contains a return value from the function *log_open*. If *lg_info* is NULL, no logging is done by the DB access methods.

DB_LOCKTAB *lk_info;
: If locking is required for the file being opened (as is the case when multiple processes or threads are accessing the same file), the *lk_info* field contains a return value from the function *lock_open*. If *lk_info* is NULL, no locking is done by the DB access methods.

If both locking and transactions are being performed (i.e., both *lk_info* and *tx_info* are non-NULL), the transaction ID will be used as the locker ID. If only locking is being performed, *db_open* will acquire a locker ID from *lock_id*(3), and will use it for all locks required for this instance of *db_open*.

DB_MPOOL *mp_info;
: If the cache for the file being opened should be maintained in a shared buffer pool, the *mp_info* field contains a return value from the function *memp_open*. If *mp_info* is NULL, a memory pool may still be created by DB, but it will be private to the application and managed by DB.

DB_TXNMGR *tx_info;

If the accesses to the file being opened should take place in the context of transactions (providing atomicity and error recovery), the *tx_info* field contains a return value from the function *txn_open* (see *db_txn*(3)). If transactions are specified, the application is responsible for making suitable calls to *txn_begin*, *txn_abort*, and *txn_commit*. If *tx_info* is NULL, no transaction support is done by the DB access methods.

When the access methods are used in conjunction with transactions, the application must abort the transaction (using *txn_abort*) if any of the transaction protected access method calls (i.e., any calls other than open, close and sync) returns a system error (e.g., deadlock, which returns EAGAIN). As described by *db_intro*(3), a system error is any value greater than 0.

DB_INFO

The access methods are configured using the DB_INFO data structure argument to *db_open*. The DB_INFO structure is typedef'd in <db.h> and has a large number of fields, most specific to a single access method, although a few are shared. The fields that are common to all access methods are listed here; those specific to an individual access method are described below. No reference to the DB_INFO structure is maintained by DB, so it is possible to discard it as soon as the *db_open* call returns.

In order to ensure compatibility with future releases of DB, all fields of the DB_INFO structure should be initialized to 0 before the structure is used. Do this by declaring the structure external or static, or by calling the C library function *bzero*(3) or *memset*(3).

If possible, defaults appropriate for the system are used for the DB_INFO fields if *dbinfo* is NULL or any fields of the DB_INFO structure are set to 0. The following DB_INFO fields may be initialized before calling *db_open*:

size_t db_cachesize;

A suggested maximum size of the memory pool cache, in bytes. If *db_cachesize* is 0, an appropriate default is used. If the *mp_info* field is also specified, this field is ignored.

Note, the minimum number of pages in the cache should be no less than 10, and the access methods will fail if an insufficiently large cache is specified. In addition, for applications that exhibit strong locality in their data access patterns, increasing the size of the cache can significantly improve application performance.

int db_lorder;

The byte order for integers in the stored database metadata. The number should represent the order as an integer, for example, big endian order is the number 4,321, and little endian order is the number 1,234. If *db_lorder* is 0, the host order of the machine where the DB library was compiled is used.

The value of *db_lorder* is ignored except when databases are being created. If a database already exists, the byte order it uses is determined when the file is read.

The access methods provide no guarantees about the byte ordering of the application data stored in the database, and applications are responsible for maintaining any necessary ordering.

size_t db_pagesize;

The size of the pages used to hold items in the database, in bytes. The minimum page size is 512 bytes and the maximum page size is 64K bytes. If *db_pagesize* is 0, a page size is selected based on the underlying filesystem I/O block size. The selected size has a lower limit of 512 bytes and an upper limit of 16K bytes.

void *(*db_malloc)(size_t);

The flag DB_DBT_MALLOC, when specified in the DBT structure, will cause the DB library to allocate memory which then becomes the responsibility of the calling application. See *db_dbt*(3) for more information.

On systems where separate heaps are maintained for applications and libraries (notably Windows NT), specifying the DB_DBT_MALLOC flag will fail because the DB library will allocate memory from a different heap than the application will use to free it. To avoid this problem, the *db_malloc* field should be set to point to the application's allocation routine. If *db_malloc* is non-NULL, it will be used to allocate the memory returned when the DB_DBT_MALLOC flag is set. The *db_malloc* function must match the calling conventions of the *malloc*(3) library routine.

BTREE

The btree data structure is a sorted, balanced tree structure storing associated key/data pairs. Searches, insertions, and deletions in the btree will all complete in O (lg base N) where base is the average number of keys per page. Often, inserting ordered data into btrees results in pages that are half-full. This implementation has been modified to make ordered (or inverse ordered) insertion the best case, resulting in nearly perfect page space utilization.

Space freed by deleting key/data pairs from the database is never reclaimed from the filesystem, although it is reused where possible. This means that the btree storage structure is grow-only. If sufficiently many keys are deleted from a tree that shrinking the underlying database file is desirable, this can be accomplished by creating a new tree from a scan of the existing one.

The following additional fields and flags may be initialized in the DB_INFO structure before calling *db_open*, when using the btree access method:

int (*bt_compare)(const DBT *, const DBT *);

The *bt_compare* function is the key comparison function. It must return an integer less than, equal to, or greater than zero if the first key argument is considered to be respectively less than, equal to, or greater than the second key argument. The same comparison function must be used on a given tree every time it is opened. If *bt_compare* is NULL, the keys are compared lexically, with shorter keys collating before longer keys.

int bt_minkey;

The minimum number of keys that will be stored on any single page. This value is used to determine which keys will be stored on overflow pages, i.e. if a key or data item is larger than the pagesize divided by the *bt_minkey* value, it will be stored on overflow pages instead of in the page itself. The *bt_minkey* value specified must be at least 2; if *bt_minkey* is 0, a value of 2 is used.

size_t (*bt_prefix)(const DBT *, const DBT *);

The *bt_prefix* function is the prefix comparison function. If specified, this function must return the number of bytes of the second key argument that are necessary to determine that it is greater than the first key argument. If the keys are equal, the key length should be returned.

This is used to compress the keys stored on the btree internal pages. The usefulness of this is data dependent, but in some data sets can produce significantly reduced tree sizes and search times. If *bt_prefix* is NULL, and no comparison function is specified, a default lexical comparison function is used. If *bt_prefix* is NULL and a comparison function is specified, no prefix comparison is done.

unsigned long flags;

The following additional flags may be specified by **or**'ing together one or more of the following values:

DB_DUP

Permit duplicate keys in the tree, i.e. insertion when the key of the key/data pair being inserted already exists in the tree will be successful. The ordering of duplicates in the tree is determined by the order of insertion, unless the ordering is otherwise specified by use of a cursor (see *db_cursor*(3) for more information.) It is an error to specify both DB_DUP and DB_RECNUM.

DB_RECNUM

Support retrieval from btrees using record numbers. For more information, see the DB_SET_RECNO flag to the *DB->get* function (below), and the cursor *c_get* function (in *db_cursor*(3)).

Logical record numbers in btrees are mutable in the face of record insertion or deletion. See the DB_RENUMBER flag in the RECNO section below for further discussion.

Maintaining record counts within a btree introduces a serious point of contention, namely the page locations where the record counts are stored. In addition, the entire tree must be locked during both insertions and deletions, effectively single-threading the tree for those operations. Specifying DB_RECNUM can result in serious performance degradation for some applications and data sets.

It is an error to specify both DB_DUP and DB_RECNUM.

HASH

The hash data structure is an extensible, dynamic hashing scheme. Backward compatible interfaces to the functions described in *dbm*(3), *ndbm*(3) and *hsearch*(3) are provided, however these interfaces are not compatible with previous file formats.

The following additional fields and flags may be initialized in the DB_INFO structure before calling *db_open*, when using the hash access method:

unsigned int h_ffactor;

The desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. The default value is 0, indicating that the fill factor will be selected dynamically as pages are filled.

u_int32_t (*h_hash)(const void *, u_int32_t);

The *h_hash* field is a user defined hash function; if *h_hash* is NULL, a default hash function is used. Since no hash function performs equally well on all possible data, the user may find that the built-in hash function performs poorly with a particular data set. User specified hash functions must take a pointer to a byte string and a length as arguments and return a u_int32_t value.

If a hash function is specified, *hash_open* will attempt to determine if the hash function specified is the same as the one with which the database was created, and will fail if it detects that it is not.

unsigned int h_nelem;

An estimate of the final size of the hash table. If not set or set too low, hash tables will expand gracefully as keys are entered, although a slight performance degradation may be noticed. The default value is 1.

unsigned long flags;

The following additional flags may be specified by **or**'ing together one or more of the following values:

DB_DUP

Permit duplicate keys in the tree, i.e. insertion when the key of the key/data pair being inserted already exists in the tree will be successful. The ordering of duplicates in the tree is determined by the order of insertion, unless the ordering is otherwise specified by use of a cursor (see *db_cursor*(3) for more information.)

RECNO

The recno access method provides support for fixed and variable length records, optionally backed by a flat text (byte stream) file. Both fixed and variable length records are accessed by their logical record number.

It is valid to create a record whose record number is more than one greater than the last record currently in the database. For example, the creation of record number 8, when records 6 and 7 do not yet exist, is not an error. However, any attempt to retrieve such records (e.g., records 6 and 7) will return DB_KEYEMPTY.

Deleting a record will not, by default, renumber records following the deleted record (see DB_RENUMBER below for more information). Any attempt to retrieve deleted records will return DB_KEYEMPTY.

The following additional fields and flags may be initialized in the DB_INFO structure before calling *db_open*, when using the recno access method:

int re_delim;

For variable length records, if the *re_source* file is specified and the DB_DELIMITER flag is set, the delimiting byte used to mark the end of a record in the source file. If the *re_source* file is specified and the DB_DELIMITER flag is not set, <newline> characters (i.e. "\n", 0x0a) are interpreted as end-of-record markers.

u_int32_t re_len;

The length of a fixed-length record.

int re_pad;

For fixed length records, if the DB_PAD flag is set, the pad character for short records. If the DB_PAD flag is not set, <space> characters (i.e., 0x20) are used for padding.

char *re_source;

The purpose of the *re_source* field is to provide fast access and modification to databases that are normally stored as flat text files.

If the *re_source* field is non-NULL, it specifies an underlying flat text database file that is read to initialize a transient record number index. In the case of variable length records, the records are separated by the byte value *re_delim*. For example, standard UNIX byte stream files can be interpreted as a sequence of variable length records separated by <newline> characters.

In addition, when cached data would normally be written back to the underlying database file (e.g., the *close* or *sync* functions are called), the in-memory copy of the database will be written back to the *re_source* file.

By default, the backing source file is read lazily, i.e., records are not read from the file until they are requested by the application. **If multiple processes (not threads) are accessing a recno database concurrently and either inserting or deleting records, the backing source file must be read in its entirety before more than a single process accesses the database, and only that process should specify the backing source file as part of the db_open call.** See the DB_SNAPSHOT flag below for more information.

Reading and writing the backing source file specified by re_source cannot be transactionally protected because it involves filesystem operations that are not part of the DB transaction methodology. For this reason, if a temporary database is used to hold the records, i.e., a NULL was specified as the *file* argument to *db_open*, it is possible to lose the contents of the *re_source* file, e.g., if the system crashes at the right instant. If a file is used to hold the database, i.e., a file name was specified as the *file* argument to *db_open*, normal database recovery on that file can be used to prevent information loss, although it is still possible that the contents of *re_source* will be lost if the system crashes.

The *re_source* file must already exist (but may be zero-length) when *db_open* is called.

For all of the above reasons, the *re_source* field is generally used to specify databases that are read-only for DB applications, and that are either generated on the fly by software tools, or modified using a different mechanism, e.g., a text editor.

unsigned long flags;

The following additional flags may be specified by **or**'ing together one or more of the following values:

DB_DELIMITER

The *re_delim* field is set.

DB_FIXEDLEN

The records are fixed-length, not byte delimited. The structure element *re_len* specifies the length of the record, and the structure element *re_pad* is used as the pad character.

Any records added to the database that are less than *re_len* bytes long are automatically padded. Any attempt to insert records into the database that are greater than *re_len* bytes long will cause the call to fail immediately and return an error.

DB_PAD
: The *re_pad* field is set.

DB_RENUMBER
: Specifying the DB_RENUMBER flag causes the logical record numbers to be mutable, and change as records are added to and deleted from the database. For example, the deletion of record number 4 causes records numbered 5 and greater to be renumbered downward by 1. If a cursor was positioned to record number 4 before the deletion, it will reference the new record number 4, if any such record exists, after the deletion. If a cursor was positioned after record number 4 before the deletion, it will be shifted downward 1 logical record, continuing to reference the same record as it did before.

Using the *c_put* or *put* interfaces to create new records will cause the creation of multiple records if the record number is more than one greater than the largest record currently in the database. For example, creating record 28, when record 25 was previously the last record in the database, will create records 26 and 27 as well as 28. Attempts to retrieve records that were created in this manner will result in an error return of DB_KEYEMPTY.

If a created record is not at the end of the database, all records following the new record will be automatically renumbered upward by 1. For example, the creation of a new record numbered 8 causes records numbered 8 and greater to be renumbered upward by 1. If a cursor was positioned to record number 8 or greater before the insertion, it will be shifted upward 1 logical record, continuing to reference the same record as it did before.

For these reasons, concurrent access to a recno database with the DB_RENUMBER flag specified may be largely meaningless, although it is supported.

DB_SNAPSHOT
: This flag specifies that any specified *re_source* file be read in its entirety when *db_open* is called. If this flag is not specified, the *re_source* file may be read lazily.

DB OPERATIONS

The DB structure returned by *db_open* describes a database type, and includes a set of functions to perform various actions, as described below. Each of these functions takes a pointer to a DB structure, and may take one or more DBT *'s and a flag value as well. The fields of the DB structure are as follows:

DBTYPE type;
: The type of the underlying access method (and file format). Set to one of DB_BTREE, DB_HASH or DB_RECNO. This field may be used to determine the type of the database after a return from *db_open* with the *type* argument set to DB_UNKNOWN.

int (*close)(DB *db, int flags);
: A pointer to a function to flush any cached information to disk, close any open cursors (see *db_cursor*(3)), free any allocated resources, and close any underlying files. Since key/data pairs are cached in memory, failing to sync the file with the *close* or *sync* function may result in inconsistent or lost information.

The *flags* parameter must be set to 0 or the following value:

DB_NOSYNC
: Do not flush cached information to disk.

The DB_NOSYNC flag is a dangerous option. It should only be set if the application is doing logging (with transactions) so that the database is recoverable after a system or application crash, or if the database is always generated from scratch after any system or application crash.

It is important to understand that flushing cached information to disk only minimizes the window of opportunity for corrupted data. While unlikely, it is possible for database corruption to happen if a system or application crash occurs while writing data to the database. To ensure that database corruption never occurs, applications must either: use transactions and logging with automatic recovery, use logging and application-specific recovery, or edit a copy of the database, and, once all applications using the database have successfully called *close*, replace the original database with the updated copy.

When multiple threads are using the DB handle concurrently, only a single thread may call the DB handle close function.

The *close* function returns the value of *errno* on failure and 0 on success.

int (*cursor)(DB *db, DB_TXN *txnid, DBC **cursorp);

A pointer to a function to create a cursor and copy a pointer to it into the memory referenced by *cursorp*.

A cursor is a structure used to provide sequential access through a database. This interface and its associated functions replaces the functionality provided by the *seq* function in previous releases of the DB library.

If the file is being accessed under transaction protection, the *txnid* parameter is a transaction ID returned from *txn_begin*, otherwise, NULL. If transaction protection is enabled, cursors must be opened and closed within the context of a transaction, and the *txnid* parameter specifies the transaction context in which the cursor may be used. See *db_cursor*(3) for more information.

The *cursor* function returns the value of *errno* on failure and 0 on success.

int (*del)(DB *db, DB_TXN *txnid, DBT *key, int flags);

A pointer to a function to remove key/data pairs from the database. The key/data pair associated with the specified *key* is discarded from the database. In the presence of duplicate key values, all records associated with the designated key will be discarded.

If the file is being accessed under transaction protection, the *txnid* parameter is a transaction ID returned from *txn_begin*, otherwise, NULL.

The *flags* parameter is currently unused, and must be set to 0.

The *del* function returns the value of *errno* on failure, 0 on success, and DB_NOTFOUND if the specified *key* did not exist in the file.

int (*fd)(DB *db, int *fdp);

A pointer to a function that copies a file descriptor representative of the underlying database into the memory referenced by *fdp*. A file descriptor referencing the same file will be returned to all processes that call *db_open* with the same *file* argument. This file descriptor may be safely used as an argument to the *fcntl*(2) and *flock*(2) locking functions. The file descriptor is not necessarily associated with any of the underlying files used by the access method.

The *fd* function only supports a coarse-grained form of locking. Applications should use the lock manager where possible.

The *fd* function returns the value of *errno* on failure and 0 on success.

int (*get)(DB *db, DB_TXN *txnid,
    DBT *key, DBT *data, int flags);

A pointer to a function that is an interface for keyed retrieval from the database. The address and length of the data associated with the specified *key* are returned in the structure referenced by *data*.

In the presence of duplicate key values, *get* will return the first data item for the designated key. Duplicates are sorted by insert order except where this order has been overridden by cursor operations. **Retrieval of duplicates requires the use of cursor operations.** See *db_cursor*(3) for details.

If the file is being accessed under transaction protection, the *txnid* parameter is a transaction ID returned from *txn_begin*, otherwise, NULL.

The *flags* parameter must be set to 0 or the following value:

DB_SET_RECNO
: Retrieve the specified numbered key/data pair from a database. Upon return, both the *key* and *data* items will have been filled in, not just the data item as is done for all other uses of the *get* function.

  The *data* field of the specified *key* must be a pointer to a memory location from which a *db_recno_t* may be read, as described in *db_dbt*(3). This memory location will be read to determine the record to be retrieved.

  For DB_SET_RECNO to be specified, the underlying database must be of type btree and it must have been created with the DB_RECNUM flag (see *db_open*(3)).

If the database is a recno database and the requested key exists, but was never explicitly created by the application or was later deleted, the *get* function returns DB_KEYEMPTY. Otherwise, if the requested key isn't in the database, the *get* function returns DB_NOTFOUND. Otherwise, the *get* function returns the value of *errno* on failure and 0 on success.

int (*put)(DB *db, DB_TXN *txnid,
        DBT *key, DBT *data, int flags);

A pointer to a function to store key/data pairs in the database. If the database supports duplicates, the *put* function adds the new data value at the end of the duplicate set.

If the file is being accessed under transaction protection, the *txnid* parameter is a transaction ID returned from *txn_begin*, otherwise, NULL.

The flags value is specified by **or**'ing together one or more of the following values:

DB_APPEND
: Append the key/data pair to the end of the database. For DB_APPEND to be specified, the underlying database must be of type recno. The record number allocated to the record is returned in the specified *key*.

DB_NOOVERWRITE
: Enter the new key/data pair only if the key does not already appear in the database.

The default behavior of the *put* function is to enter the new key/data pair, replacing any previously existing key if duplicates are disallowed, or to add a duplicate entry if duplicates are allowed. Even if the designated database allows duplicates, a call to *put* with the DB_NOOVERWRITE flag set will fail if the key already exists in the database.

The *put* function returns the value of *errno* on failure, 0 on success, and DB_KEYEXIST if the DB_NOOVERWRITE *flag* was set and the key already exists in the file.

int (*sync)(DB *db, int flags);

A pointer to a function to flush any cached information to disk. If the database is in memory only, the *sync* function has no effect and will always succeed.

The *flags* parameter is currently unused, and must be set to 0.

See the *close* function description above for a discussion of DB and cached data.

The *sync* function returns the value of *errno* on failure and 0 on success.

int (*stat)(DB *db, void *sp,
        void *(*db_malloc)(size_t), int flags);

A pointer to a function to create a statistical structure and copy a pointer to it into user-specified memory locations. Specifically, if *sp* is non-NULL, a pointer to the statistics for the database are copied into the memory location it references.

Statistical structures are created in allocated memory. If *db_malloc* is non-NULL, it is called to allocate the memory, otherwise, the library function *malloc*(3) is used. The function *db_malloc* must match the calling conventions of the *malloc*(3) library routine. Regardless, the caller is responsible for deallocating the returned memory. To deallocate the returned memory, free each returned memory pointer; pointers inside the memory do not need to be individually freed.

In the presence of multiple threads or processes accessing an active database, the returned information may be out-of-date.

This function may access all of the pages in the database, and therefore may incur a severe performance penalty and have obvious negative effects on the underlying buffer pool.

The *flags* parameter must be set to 0 or the following value:

DB_RECORDCOUNT
: Fill in the *bt_nrecs* field of the statistics structure, but do not collect any other information. This flag makes it reasonable for applications to request a record count from a database without incurring a performance penalty. It is only available for recno databases, or btree databases where the underlying database was created with the DB_RECNUM flag.

The *stat* function returns the value of *errno* on failure and 0 on success.

In the case of a btree or recno database, the statistics are stored in a structure of type DB_BTREE_STAT (typedef'd in <db.h>). The following fields will be filled in:

u_int32_t bt_magic;
: Magic number that identifies the file as a btree file.

u_int32_t bt_version;
: The version of the btree file type.

u_int32_t bt_flags;
: Permanent database flags, including DB_DUP, DB_FIXEDLEN, DB_RECNUM and DB_RENUMBER.

u_int32_t bt_minkey;
: The *bt_minkey* value specified to *db_open*(3), if any.

u_int32_t bt_re_len;
: The *re_len* value specified to *db_open*(3), if any.

u_int32_t bt_re_pad;
: The *re_pad* value specified to *db_open*(3), if any.

u_int32_t bt_pagesize;
: Underlying tree page size.

u_int32_t bt_levels;
: Number of levels in the tree.

u_int32_t bt_nrecs;
: Number of data items in the tree (since there may be multiple data items per key, this number may not be the same as the number of keys).

u_int32_t bt_int_pg;
: Number of tree internal pages.

u_int32_t bt_leaf_pg;
: Number of tree leaf pages.

u_int32_t bt_dup_pg;
: Number of tree duplicate pages.

u_int32_t bt_over_pg;
: Number of tree overflow pages.

u_int32_t bt_free;
: Number of pages on the free list.

u_int32_t bt_freed;
: Number of pages made available for reuse because they were emptied.

u_int32_t bt_int_pgfree;
: Number of bytes free in tree internal pages.

u_int32_t bt_leaf_pgfree;
: Number of bytes free in tree leaf pages.

u_int32_t bt_dup_pgfree;
: Number of bytes free in tree duplicate pages.

u_int32_t bt_over_pgfree;
: Number of bytes free in tree overflow pages.

u_int32_t bt_pfxsaved;
: Number of bytes saved by prefix compression.

u_int32_t bt_split;
: Total number of tree page splits (includes fast and root splits).

u_int32_t bt_rootsplit;
: Number of root page splits.

u_int32_t bt_fastsplit;
: Number of fast splits. When sorted keys are added to the database, the DB btree implementation will split left or right to increase the page-fill factor. This number is a measure of how often it was possible to make such a split.

u_int32_t bt_added;
: Number of keys added.

u_int32_t bt_deleted;
: Number of keys deleted.

u_int32_t bt_get;
: Number of keys retrieved. (Note, this value will not reflect any keys retrieved when the database was open for read-only access, as there is no permanent location to store the information in this case.)

u_int32_t bt_cache_hit;
: Number of hits in tree fast-insert code. When sorted keys are added to the database, the DB btree implementation will check the last page where an insert occurred before doing a full lookup. This number is a measure of how often the lookup was successful.

u_int32_t bt_cache_miss;
: Number of misses in tree fast-insert code. See the description of bt_cache_hit; this number is a measure of how often the lookup failed.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *db_open*:

DB_HOME
: If the *dbenv* argument to *db_open* was initialized using *db_appinit*, the environment variable DB_HOME may be used as the path of the database home for the interpretation of the *dir* argument to *db_open*, as described in *db_appinit*(3). Specifically, *db_open* is affected by the configuration string value of DB_DATA_DIR.

EXAMPLES

Applications that create short-lived databases that are discarded or recreated when the system fails and are unconcerned with concurrent access and loss of data due to catastrophic failure, may wish to use the *db_open* functionality without other parts of the DB library. Such applications will only be concerned with the DB access methods. The DB access methods will use the memory pool subsystem, but the application is unlikely to be aware of this. See the files *example/ex_access.c* and *example/ex_btrec.c* in the DB source distribution for C language code examples of how such applications might use the DB library.

ERRORS

The *db_open* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DB->sync(3), calloc(3), close(2), fcntl(2), fflush(3), lock_get(3), lock_id(3), lock_put(3), lock_vec(3), log_put(3), log_register(3), log_unregister(3), malloc(3), memcpy(3), memmove(3), memp_close(3), memp_fclose(3), memp_fget(3), memp_fopen(3), memp_fput(3), memp_fset(3), memp_fsync(3), memp_open(3), memp_register(3), memset(3), mmap(2), munmap(2), open(2), read(2), realloc(3), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), time(3), and unlink(2).

In addition, the *db_open* function may fail and return *errno* for the following conditions:

[EAGAIN]
: A lock was unavailable.

[EINVAL]
: An invalid flag value or parameter was specified (e.g., unknown database type, page size, hash function, recno pad byte, byte order) or a flag value or parameter that is incompatible with the current *file* specification.

  The DB_THREAD flag was specified and spinlocks are not implemented for this architecture.

  There is a mismatch between the version number of *file* and the software.

  A *re_source* file was specified with either the DB_THREAD flag or a non-NULL *tx_info* field in the DB_ENV argument to db_open.

[ENOENT]
: A non-existent *re_source* file was specified.

[EPERM]
: Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

The *DB->close* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DB->sync(3), calloc(3), close(2), fflush(3), lock_get(3), lock_put(3), lock_vec(3), log_put(3), log_unregister(3), malloc(3), memcpy(3), memmove(3), memp_close(3), memp_fclose(3), memp_fget(3), memp_fput(3), memp_fset(3), memp_fsync(3), memset(3), munmap(2), realloc(3), and strerror(3).

The *DB->cursor* function may fail and return *errno* for any of the errors specified for the following DB and library functions: calloc(3).

In addition, the *DB->cursor* function may fail and return *errno* for the following conditions:

[EINVAL]
: An invalid flag value or parameter was specified.

[EPERM]
: Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

The *DB->del* function may fail and return *errno* for any of the errors specified for the following DB and library functions: calloc(3), fcntl(2), fflush(3), lock_get(3), lock_id(3), lock_put(3), lock_vec(3), log_put(3), malloc(3), memcmp(3), memcpy(3), memmove(3), memp_fget(3), memp_fput(3), memp_fset(3), memset(3), realloc(3), and strerror(3).

In addition, the *DB->del* function may fail and return *errno* for the following conditions:

[EAGAIN]
: A lock was unavailable.

[EINVAL]
: An invalid flag value or parameter was specified.

[EPERM]
: Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

In addition, the *DB->fd* function may fail and return *errno* for the following conditions:

[ENOENT]
The *DB->fd* function was called for an in-memory database, or no underlying file has yet been created.

[EPERM]
Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

The *DB->get* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBcursor->c_get(3), calloc(3), fcntl(2), fflush(3), lock_get(3), lock_id(3), lock_put(3), lock_vec(3), log_put(3), malloc(3), memcmp(3), memcpy(3), memmove(3), memp_fget(3), memp_fput(3), memp_fset(3), memset(3), realloc(3), and strerror(3).

In addition, the *DB->get* function may fail and return *errno* for the following conditions:

[EAGAIN]
A lock was unavailable.

[EINVAL]
An invalid flag value or parameter was specified.

The DB_THREAD flag was specified to the *db_open*(3) function and neither the DB_DBT_MALLOC or DB_DBT_USERMEM flags were set in the DBT.

A record number of 0 was specified.

[EPERM]
Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

The *DB->put* function may fail and return *errno* for any of the errors specified for the following DB and library functions: calloc(3), fcntl(2), fflush(3), lock_get(3), lock_id(3), lock_put(3), lock_vec(3), log_put(3), malloc(3), memcmp(3), memcpy(3), memmove(3), memp_fget(3), memp_fput(3), memp_fset(3), memset(3), realloc(3), and strerror(3).

In addition, the *DB->put* function may fail and return *errno* for the following conditions:

[EACCES]
An attempt was made to modify a read-only database.

[EAGAIN]
A lock was unavailable.

[EINVAL]
An invalid flag value or parameter was specified.

A record number of 0 was specified.

An attempt was made to add a record to a fixed-length database that was too large to fit.

An attempt was made to do a partial put.

[EPERM]
Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

[ENOSPC]
A btree exceeded the maximum btree depth (255).

The *DB->stat* function may fail and return *errno* for any of the errors specified for the following DB and library functions: calloc(3), fcntl(2), fflush(3), lock_get(3), lock_id(3), lock_put(3), lock_vec(3), malloc(3), memcpy(3), memp_fget(3), memp_fput(3), and memset(3).

The *DB->sync* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DB->get(3), DB->sync(3), calloc(3), close(2), fcntl(2), fflush(3), lock_get(3), lock_id(3), lock_put(3), lock_vec(3), log_put(3), malloc(3), memcpy(3), memmove(3), memp_fget(3), memp_fput(3), memp_fset(3), memp_fsync(3), memset(3), munmap(2), open(2), realloc(3), strerror(3), unlink(2), and write(2).

In addition, the *DB->sync* function may fail and return *errno* for the following conditions:

[EINVAL]
An invalid flag value or parameter was specified.

[EPERM]
Database corruption was detected. All subsequent database calls (other than *DB->close*) will return EPERM.

SEE ALSO

The Ubiquitous B-tree, Douglas Comer, ACM Comput. Surv. 11, 2 (June 1979), 121-138.

Prefix B-trees, Bayer and Unterauer, ACM Transactions on Database Systems, Vol. 2, 1 (March 1977), 11-26.

The Art of Computer Programming Vol. 3: Sorting and Searching, D.E. Knuth, 1968, pp 471-480.

Dynamic Hash Tables, Per-Ake Larson, Communications of the ACM, April 1988.

A New Hash Package for UNIX, Margo Seltzer, USENIX Proceedings, Winter 1991.

Document Processing in a Relational Database System, Michael Stonebraker, Heidi Stettner, Joseph Kalash, Antonin Guttman, Nadene Lynn, Memorandum No. UCB/ERL M82/32, May 1982.

db_archive(1), *db_checkpoint*(1), *db_deadlock*(1), *db_dump*(1), *db_load*(1), *db_recover*(1), *db_stat*(1), *db_intro*(3), *db_appinit*(3), *db_cursor*(3), *db_dbm*(3), *db_internal*(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_thread*(3), *db_txn*(3)