NAME

db_btree − btree database access method

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 transaction, locking, logging and file buffering functionality, as well as various indexed access methods. Many of the functional groups (e.g. the memory pool functions) are useful independently of the rest of the DB functions, although some functional groups are explicitly based on other functional groups (e.g. transactions and logging). For a general description of transactions, see *db_txn*(3). For a general description of the access methods, see *db_open*(3) and then the individual access method manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3) and *db_recno*(3). For a general description of the lock manager, see *db_lock*(3). For a general description of the memory pool manager, see *db_mpool*(3).

This manual page describes specific details of the btree access method.

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 fill factor. Often, inserting ordered data into btrees results in a low fill factor. This implementation has been modified to make ordered insertion the best case, resulting in a much better than normal page fill factor.

ACCESS METHOD SPECIFIC INFORMATION

The btree access method specific data structure provided to *db_open* is typedef'd and named BTREEINFO. A BTREEINFO structure has at least the following fields, which may be initialized before calling *db_open*:

u_int cachesize;

A suggested maximum size (in bytes) of the memory cache. This value is **only** advisory, and the access method will allocate more memory rather than fail. Since every search examines the root page of the tree, caching the most recently used pages substantially improves access time. In addition, physical writes are delayed as long as possible, so a moderate cache can reduce the number of I/O operations significantly. Obviously, using a cache increases (but only increases) the likelihood of corruption or lost data if the system crashes while a tree is being modified. If *cachesize* is 0 (no size is specified) a default cache is used.

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

Compare 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 *compare* is NULL (no comparison function is specified), the keys are compared lexically, with shorter keys considered less than longer keys.

u_long flags;

The flag value is specified by *or*'ing any of the following values:

R_DUP

Permit duplicate keys in the tree, i.e. permit insertion if the key to be inserted already exists in the tree. The default behavior, as described in *db_open*(3), is to overwrite a matching key when inserting a new key or to fail if the R_NOOVERWRITE flag is specified. The R_DUP flag is overridden by the R_NOOVERWRITE flag, and if the R_NOOVERWRITE flag is specified, attempts to insert duplicate keys into the tree will fail.

If the database contains duplicate keys, the order of retrieval of key/data pairs is undefined if the *get* function is used, however, *seq* function calls with the R_CURSOR flag set will always return the logical ‘‘first’’ of any group of duplicate keys.

int 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 would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used.

int maxkeypage;

The maximum number of keys which will be stored on any single page. This functionality is not currently implemented.

int minkeypage;

The minimum number of keys which 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 longer than the pagesize divided by the minkeypage value, it will be stored on overflow pages instead of in the page itself. If *minkeypage* is 0 (no minimum number of keys is specified) a value of 2 is used.

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

Prefix is the prefix comparison function. If specified, this function must return the number of bytes of the second key argument which are necessary to determine that it is greater than the first key argument. If the keys are equal, the key length should be returned. Note, the usefulness of this function is very data dependent, but, in some data sets can produce significantly reduced tree sizes and search times. If *prefix* is NULL (no prefix function is specified), **and** no comparison function is specified, a default lexical comparison function is used. If *prefix* is NULL and a comparison function is specified, no prefix comparison is done.

u_int psize;

Page size is the size (in bytes) of the pages used for nodes in the tree. The minimum page size is 512 bytes and the maximum page size is 64K. If *psize* is 0 (no page size is specified) a page size is chosen based on the underlying file system I/O block size.

If the file already exists (and the O_TRUNC flag is not specified), the values specified for the parameters flags, lorder and psize are ignored in favor of the values used when the tree was created.

DB OPERATIONS

The functions returned by *db_open* for the btree access method are as described in *db_open*(3), with the following exceptions and additions:

type The type is DB_BTREE.

del Space freed up by deleting key/data pairs from the tree is never reclaimed, although it is reused where possible. This means that the btree storage structure is grow-only. The only solutions are to avoid excessive deletions, or to create a fresh tree periodically from a scan of an existing one.

put The *put* function takes the following additional flags:

R_SETCURSOR

Store the key/data pair, setting or initializing the position of the cursor to reference it.

seq Forward sequential scans of a tree are from the least key to the greatest.

The returned key for the *seq* function is not necessarily an exact match for the specified key in the btree access method. The returned key is the smallest key greater than or equal to the specified key, permitting partial key matches and range searches.

The *seq* function takes the following additional flags:

R_LAST

The last key/data pair of the database is returned, and the cursor is set or initialized to reference it.

R_PREV
: Retrieve the key/data pair immediately before the cursor. If the cursor is not yet set, this is the same as the R_LAST flag.

ERRORS

The *btree* access method functions may fail and set *errno* for any of the errors specified for the library function *db_open*(3).

SEE ALSO

db_hash(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_recno*(3), *db_txn*(3)

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.

NAME

db_hash − hash database access method

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 transaction, locking, logging and file buffering functionality, as well as various indexed access methods. Many of the functional groups (e.g. the memory pool functions) are useful independently of the rest of the DB functions, although some functional groups are explicitly based on other functional groups (e.g. transactions and logging). For a general description of transactions, see *db_txn*(3). For a general description of the access methods, see *db_open*(3) and then the individual access method manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3) and *db_recno*(3). For a general description of the lock manager, see *db_lock*(3). For a general description of the memory pool manager, see *db_mpool*(3).

This manual page describes specific details of the hashing access method.

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

ACCESS METHOD SPECIFIC INFORMATION

The hash access method specific data structure provided to *db_open* is typedef'd and named HASHINFO. A HASHINFO structure has at least the following fields, which may be initialized before calling *db_open*:

u_int bsize;

Bsize defines the hash table bucket size, and is, by default, 256 bytes. It may be preferable to increase the page size for disk-resident tables and tables with large data items.

u_int cachesize;

A suggested maximum size, in bytes, of the memory cache. This value is **only** advisory, and the access method will allocate more memory rather than fail.

u_int ffactor;

Ffactor indicates a 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 8.

u_int32_t (*hash)(const void *, size_t);

Hash is a user defined hash function. Since no hash function performs equally well on all possible data, the user may find that the built-in hash function does poorly on a particular data set. User specified hash functions must take two arguments (a pointer to a byte string and a length) and return a 32-bit quantity to be used as the hash 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 is not.

int 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 would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used. If the file already exists, the specified value is ignored and the value specified when the tree was created is used.

u_int nelem;

Nelem is 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.

If the file already exists (and the O_TRUNC flag is not specified), the values specified for the parameters bsize, ffactor, lorder and nelem are ignored and the values specified when the tree was created are used.

DB OPERATIONS

The functions returned by *db_open* for the hash access method are as described in *db_open*(3).

ERRORS

The *hash* access method functions may fail and set *errno* for any of the errors specified for the library function *db_open*(3).

SEE ALSO

db_btree(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_recno*(3), *db_txn*(3)

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

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

NAME

db_lock − general purpose lock manager

SYNOPSIS

```
#include <db_lock.h>

int
lock_create(const char *path, mode_t mode,
        int lock_modes, const int8_t conflicts[][], u_int maxlocks);

LOCK_TABLE_T *
lock_open(const char *path);

int
lock_vec(LOCK_TABLE_T *lt, DBT *locker, struct timespec *timeout,
        LOCK_REQ_T list[], int nlist, LOCK_REQ_T **elistp, DBT *conflict);

int
lock_get(LOCK_TABLE_T *lt, const DBT *locker,
        const DBT *obj, const lock_mode_t lock_mode, LOCK_T **lockp);

int
lock_put(LOCK_T *lockp);

int
lock_close(LOCK_TABLE_T *lt);

int
lock_unlink(const char *path, int force);
```

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 transaction, locking, logging and file buffering functionality, as well as various indexed access methods. Many of the functional groups (e.g. the memory pool functions) are useful independently of the rest of the DB functions, although some functional groups are explicitly based on other functional groups (e.g. transactions and logging). For a general description of transactions, see *db_txn*(3). For a general description of the access methods, see *db_open*(3) and then the individual access method manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3) and *db_recno*(3). For a general description of the lock manager, see *db_lock*(3). For a general description of the memory pool manager, see *db_mpool*(3).

This manual page describes specific details of the locking interface.

Db_lock is the library interface intended to provide general-purpose locking. While designed to work with the other DB functions, these functions are also useful for more general locking purposes. Locks can be shared between processes.

The function *lock_create* creates and initializes the lock table identified by the *path* directory. This directory must already exist when *lock_create* is called. If the lock table identified by *path* already exists, then *lock_create* returns success without further action. The files associated with the lock table are created in the directory specified by *path*. (The group of the created files is based on the system and directory defaults, and is not further specified by *lock_create*.) All files created by *lock_create* are created with mode *mode* (as described in *chmod*(2)) and modified by the process' umask value (see *umask*(2)).

The parameter *lock_modes* is the number of lock modes to be recognized by the lock table (including the ''not-granted'' mode). The parameter *conflicts* is an *lock_modes* by *lock_modes* array. A non-0 value for:

conflicts[requested_mode][held_mode]

indicates that *requested_mode* and *held_mode* conflict. The ''not-granted'' mode must be represented by 0.

The include file <db_lock.h> declares two commonly used conflict arrays:

int lock_sx_n;
const int8_t lock_sx_c[lock_sx_n][lock_sx_n];
These variables specify a conflict array for a simple scheme using shared and exclusive lock modes.

int lock_g_n;
const int8_t lock_g_c[lock_g_n][lock_g_n];
These variables specify a conflict array that involves various intent lock modes (e.g. intent shared) that are used for multigranularity locking.

In addition, <db_lock.h> defines the following macros that name lock modes for use with the standard tables above:

LOCK_IS
intent shared
LOCK_IX
intent exclusive
LOCK_NG
not granted (always 0)
LOCK_S
shared
LOCK_SIX
shared/intent exclusive
LOCK_X
exclusive

Maxlocks is the maximum number of locks to be held or requested in the table, and is used by *lock_create* to estimate how much space to allocate for various lock-table data structures.

The function *lock_create* returns -1 on failure, setting *errno*, and 0 on success.

The function *lock_open* returns a pointer to the lock table identified by *path*, which must have already been created by a call to *lock_create*. The process must have permission to read and write files with owners, groups and permissions as described for *lock_create*. The *lock_open* function returns NULL on failure, setting *errno*.

The function *lock_vec* atomically obtains and releases one or more locks from the designated table. The function *lock_vec* is intended to support acquisition or trading of multiple locks under one lock table semaphore, as is needed for lock coupling or in multigranularity locking for lock escalation.

If any of the requested locks cannot be acquired or any of the locks to be released cannot be released, no locks are acquired and no locks are released, and *lock_vec* returns an error. The function *lock_vec* returns 0 on success. If an error occurs, *lock_vec* returns one of the following values. In addition, if *elistp* is not NULL, it is set to point to the LOCK_REQ_T entry which was being processed when the error occurred.

LOCK_GET_DEADLOCK
The specified *locker* was selected as a victim in order to resolve a deadlock. In this case, if the *conflict* argument is non-NULL, it is set to reference the identity of a locker holding the lock referenced by *elistp* at the time the request was denied. (This identity resides in static memory and may be overwritten by subsequent calls to *lock_vec*).

LOCK_GET_ERROR
An error occurred and the external variable *errno* has been set to indicate the error.

LOCK_GET_NOTHELD
The lock cannot be released, as it was not held by the *locker*.

LOCK_GET_RESOURCE
The lock manager is unable to grant the requested locks because of limited internal resources. (Releasing locks may allow future calls to *lock_vec* to succeed.)

LOCK_GET_TIMEOUT
A timeout argument was specified, and the requested locks were not available soon enough. In this case, if the *conflict* argument is non-NULL, it is set to reference the identity of a locker holding the lock referenced by *elistp* at the time the request was denied. (This identity resides in static memory and may be overwritten by subsequent calls to *lock_vec*).

The *locker* argument specified to *lock_vec* is a pointer to an untyped byte string which identifies the entity requesting or releasing the lock. If *locker* is NULL, the calling process' pid is used instead.

The *timeout* argument provided to *lock_vec* specifies a maximum interval to wait for the locks to be granted. If *timeout* is NULL, it is ignored, and *lock_vec* will not return until all of the locks are acquired or an error has occurred.

The *list* array provided to *lock_vec* is typedef'd in <db_lock.h> as LOCK_REQ_T. A LOCK_REQ_T structure has at least the following fields, which must be initialized before calling *lock_vec*:

enum lockop op;
The operation to be performed, which must be set to one of the following values:

LOCK_GET
Get a lock, as defined by the values of *locker*, *obj* and *lock_mode*. Upon return from *lock_vec*, if the *lockp* field is non-NULL, a reference to the acquired lock is stored there. (This reference is invalidated by any call to *lock_vec* or *lock_put* which releases the lock.)

LOCK_PUT
The lock referenced by the contents of the *lockp* field is released.

LOCK_PUT_ALL
All locks held by the *locker* are released. (Any locks acquired as a part of the current call to *lock_vec* are not considered for this operation).

LOCK_PUT_OBJ
All locks held by the *locker*, on the object *obj*, with the mode specified by *lock_mode*, are released. A *lock_mode* of LOCK_NG indicates that all locks on the object should be released. (Any locks acquired as a part of the current call to *lock_vec* are not considered for this operation).

const DBT obj;
An untyped byte string which specifies the object to be locked or released.

const lock_mode_t lock_mode;
The lock mode, used as an index into *lt*'s conflict array.

LOCK_T **lockp;
A pointer to a pointer to a lock reference.

The *nlist* argument specifies the number of elements in the *list* array.

The function *lock_get* is a simple interface to the *lock_vec* functionality, and is equivalent to calling the *lock_vec* function with the *lt* and *locker* arguments, NULL *timeout*, *elistp* and *conflict* arguments, and a single element *list* array, for which the *op* field is LOCK_GET, and the *obj*, *lock_mode* and *lockp* fields are represented by the arguments of the same name. Note that the type of the *obj* argument to *lock_get* is different from the *obj* element found in the LOCK_REQ_T structure. The *lock_get* function returns success and failure as described for the *lock_vec* function.

The function *lock_put* is a simple interface to the *lock_vec* functionality, and is equivalent to calling the *lock_vec* function with a single element *list* array, for which the *op* field is LOCK_PUT and the *lockp* field is represented by the argument of the same name. Note that the type of the *lockp* argument to *lock_put* is different from the *lockp* element found in the LOCK_REQ_T structure. The *lock_put* function returns success and failure as described for the *lock_vec* function.

The function *lock_close* disassociates the calling process from the lock table *lt*, after releasing all locks held or requested by that process. The function *lock_close* returns -1 on failure, setting *errno*, and 0 on success.

The function *lock_unlink* destroys the lock table identified by the directory *path*, removing all files used to implement the lock table. (The directory *path* is not removed.) If there are processes which have called *lock_open* without calling *lock_close* (i.e., there are processes currently using the lock table), *lock_unlink* will fail without further action, unless the force flag is set, in which case *lock_unlink* will attempt to delete the lock table files regardless of any processes still using the lock table. Any accesses to a removed lock table will likely result in unexpected behavior. The function *lock_unlink* returns -1 on failure, setting *errno*, and 0 on success.

In the case of catastrophic or system failure, it is possible to clean up a lock table by removing all of the files in the directory specified to the *lock_create* function, as lock table files are never created in any directory other than the one specified to *lock_create*.

ERRORS

The *lock_create* function may fail and set *errno* for any of the errors specified for the library routines *mmap*(2), *open*(2) and *malloc*(3).

The *lock_open* function may fail and set *errno* for any of the errors specified for the library routine *mmap*(2) and *open*(2).

The *lock_close* function may fail and set *errno* for any of the errors specified for the library routine *close*(2) and *munmap*(2).

The *lock_unlink* function may fail and set *errno* for any of the errors specified for the library function *unlink*(2) or the following:

[EBUSY]
The lock table was in use and the force flag was not set.

SEE ALSO

db_btree(3), *db_hash*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_recno*(3), *db_txn*(3)

BUGS

The *maxlocks* parameter is a kluge, and should be deleted in favor of dynamically expanding the lock table.

NAME

db_log – log-manager access method

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 transaction, locking, logging and file buffering functionality, as well as various indexed access methods. Many of the functional groups (e.g. the memory pool functions) are useful independently of the rest of the DB functions, although some functional groups are explicitly based on other functional groups (e.g. transactions and logging). For a general description of transactions, see *db_txn*(3). For a general description of the access methods, see *db_open*(3) and then the individual access method manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3) and *db_recno*(3). For a general description of the lock manager, see *db_lock*(3). For a general description of the memory pool manager, see *db_mpool*(3).

This manual page describes specific details of the logging access method.

These functions provide a general-purpose logging facility sufficient for transaction management. Logs can be shared by multiple processes.

A log is represented by the directory, *not the file*, named by the first argument to *db_open*(3). The first argument must be non-NULL, and the directory must already exist *db_open* is called. In that directory, the log is stored in one or more files named in the format "log.YYYY.MM.DD.HH.MM.SS", where "YYYY.MM.DD.HH.SS" is the approximate creation time of the log file, and is guaranteed to be unique in the directory.

The group of the created files is based on the system and directory defaults, and is not further specified by the log access method. All files are created with the *mode* specified to *db_open*, (as described in *chmod*(2)) and modified by the process' umask value (see *umask*(2)).

The *flags* argument to *db_open* must be 0 for the *db_log* access method.

ACCESS METHOD SPECIFIC INFORMATION

The log access method specific data structure provided to *db_open* is typedef'd and named LOGINFO. A LOGINFO structure has at least the following fields, which may be initialized before calling *db_open*:

off_t max_file_size;

The maximum size of a single file in the log. If not specified, the maximum size defaults to an implementation-specific value.

int 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 would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used.

If the log already exists, the values specified for the parameters max_file_size and lorder are ignored in favor of the values used when the log was created.

DB OPERATIONS

The data part of the key/data pair used by the log access method is the same as for other access methods. The key is different. Each log record is identified by a log sequence number (LSN), which is stored in a DBT, and which is used as the *key* for all log functions that take *key* arguments. Applications cannot create LSN's, and all LSN's provided to functions as arguments must first be retrieved using the *put* or *seq* functions. To provide a distinguished value for applications, it is guaranteed that no valid LSN will ever have a size of 0.

Applications can compare LSN's using the *log_lsn_compare* function (see below).

Applications can associate LSN's with specific log files. The function *log_lsn_file* (see below), returns the name of the log file containing the record with a specified LSN. (The mapping of LSN to file is needed for database administration. For example, a transaction manager typically records the earliest LSN needed for restart, and the database administrator may want to archive log files to tape when they contain only LSN's before the earliest one needed for restart.)

Applications can truncate the log file up to a specific LSN using the *log_trunc* function (see below).

The functions returned by *db_open* for the log access method are as described in *db_open*, with the following exceptions and additions:

type The type is DB_LOG.

del The *del* function always returns an error for the log-manager access method, setting *errno* to EINVAL.

int (*log_flush)(const DB *db, const DBT *lsn);
The *log_flush* function flushes the log up to and including the log record *lsn*. The function *log_flush* returns -1 on failure, setting *errno*, and 0 on success.

int (*log_lsn_compare)(const DB *,
const DBT *lsn1, const DBT *lsn2); A pointer to a function which is provided to permit applications to compare LSN's. The *log_lsn_compare* function returns an integer less than, equal to, or greater than zero if the first LSN is considered to be respectively less than, equal to, or greater than the second LSN.

int (*log_lsn_file)(const DB *db,
const DBT *lsn, char *name);
The *log_lsn_file* function stores a pointer to the name of the file containing *lsn* in the address referenced by *name.* This pointer is to an internal static object, and subsequent calls to the same function will modify the same object.

The function *log_lsn_file* returns -1 on failure, setting *errno*, and 0 on success.

int (*log_unlink)(const char *path, int force);
The *log_unlink* function destroys the log represented by *path*. If the *force* parameter is not set to 1 and there are other processes using the log, then *log_unlink* will return -1, setting *errno* to EBUSY. If *force is not set or there are no processes using the log, then all files* used by the log are destroyed. *log_unlink* will return -1 on failure, setting *errno*, and 0 on success.

int (*log_trunc)(const DB *db, const DBT *lsn);
The *log_trunc* function truncates the log up to an LSN which is less than *lsn*. The function *log_trunc* returns -1 on failure, setting *errno*, and 0 on success.

put A log record containing *data* is appended to the log. Unlike the *put* functions for other access methods, the key parameter is not initialized by the application, instead, the LSN assigned to the data is returned in the *key* parameter.

The caller is responsible for providing any necessary structure to *data* . (For example, in a write-ahead logging protocol, the application must understand what part of *data* is an operation code, what part is redo information, and what part is undo information. In addition, most transaction managers will store in *data* the LSN of the previous log record for the same transaction, to support chaining back through the transaction's log records during undo.)

The parameter *flag* must be set to 0 or exactly one of the following values:

R_CHECKPOINT
Specify the key/data pair of the current call as the one to be returned when the *seq* function is next called with the R_CHECKPOINT flag.

R_FLUSH
: Flush immediately (ignoring any possibility for group commit).

seq
: The *seq* function takes the following additional flag:

R_CHECKPOINT
: The last key/data pair stored by the *put* function (using the R_CHECKPOINT flag) is returned, and the cursor is set or initialized to reference it. The expected use of this flag is during restart and to determine what part of the log must be available for restart. Therefore, the log record retrieved with R_CHECKPOINT should contain all the information that the transaction manager will need for this purpose.

sync
: The *sync* function always returns an error for the log-manager access method, setting *errno* to EINVAL.

SEE ALSO

db_btree(3), *db_hash*(3), *db_lock*(3), *db_mpool*(3), *db_open*(3), *db_recno*(3), *db_txn*(3)

NAME

db_mpool – general purpose shared memory buffer pool

SYNOPSIS

```
#include <db.h>
#include <mpool.h>

int
mpool_create(char *path, mode_t mode, size_t cachesize, u_long flags);

MPOOL *
mpool_open(char *path);

int
mpool_close(MPOOL *mp);

MPOOLFILE *
mpool_fopen(MPOOL *mp, char *path, size_t pagesize, void *pgcookie,
        int (*pgin)(MPOOLFILE *mpf,
            pgno_t pgno, void *pgaddr, void *pgcookie),
        int (*pgout)(MPOOLFILE *mpf,
            pgno_t pgno, void *pgaddr, void *pgcookie);

int
mpool_fclose(MPOOLFILE *mpf);

void *
mpool_get(MPOOLFILE *mpf, pgno_t *pgnoaddr, u_long flags,
        int (*callback)(MPOOLFILE *mpf, pgno_t pgno));

int
mpool_put(MPOOLFILE *mpf, void *pgaddr, u_long flags);

int
mpool_sync(MPOOLFILE *mpf);

int
mpool_unlink(const char *path, int force);

void
mpool_stat(MPOOL *mp, FILE *fp);
```

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 transaction, locking, logging and file buffering functionality, as well as various indexed access methods. Many of the functional groups (e.g. the memory pool functions) are useful independently of the rest of the DB functions, although some functional groups are explicitly based on other functional groups (e.g. transactions and logging). For a general description of transactions, see *db_txn*(3). For a general description of the access methods, see *db_open*(3) and then the individual access method manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3) and *db_recno*(3). For a general description of the lock manager, see *db_lock*(3). For a general description of the memory pool manager, see *db_mpool*(3).

This manual page describes specific details of the memory pool interface.

The *db_mpool* function is the library interface intended to provide general-purpose, page-oriented buffer management of one or more files. While designed to work with the other DB functions, these functions are also useful for more general purposes. The memory pools (MPOOL's) are referred to in this document as simply ‘‘pools’’. Pools may be shared between processes. Pools are usually filled by pages from one or more files (MPOOLFILE's). Pages in the pool are replaced in LRU (least-recently-used) order, with each new page replacing the page which has been unused the longest. Pages retrieved from the pool using *mpool_get* are ‘‘pinned’’ in memory, by default, until they are returned to the pool using the *mpool_put* function.

The function *mpool_create* creates and initializes the memory pool identified by the *path* directory. This directory must already exist when *mpool_create* is called. If the memory pool identified by *path* already exists, then *mpool_create* returns success without further action. The files associated with the memory pool are created in the directory specified by *path*. (The group of the created files is based on the system and directory defaults, and is not further specified by *mpool_create*.) All files created by *mpool_create* are created with mode *mode* (as described in *chmod*(2)) and modified by the process' umask value (see *umask*(2)).

The *cachesize* argument specifies the size of the pool in bytes, and should be the size of the normal working set of the application with some small amount of additional memory for unusual situations. If the number of bytes currently ‘‘pinned’’ in memory exceeds *cachesize*, the *db_mpool* functions will attempt to allocate more memory and do not necessarily fail, although they may suffer performance degradation.

The *flags* argument is set by *or*'ing any of the following values:

MPOOL_PRIVATE
: The pool is not shared by other processes or threads, so no locking of pool resources is required.

The function *mpool_open* returns a pointer to the memory pool identified by *path*, which must have already been created by a call to *mpool_create*. The process must have permission to read and write files with owners, groups and permissions as described for *mpool_create*. The *mpool_open* function returns NULL on failure, setting *errno*.

The *mpool_close* function closes the pool indicated by the MPOOL pointer *mp*, as returned by *mpool_open*. This function does **not** imply a call to *mpool_sync* (or to *mpool_fclose*) i.e. no pages are written to the source file as as a result of calling *mpool_close*. The function *mpool_close* returns -1 on failure, setting *errno*, and 0 on success.

The function *mpool_fopen* opens a file for buffering in the pool specified by the MPOOL argument. The *path* argument is the name of the file to be opened. The *pagesize* argument is the size, in bytes, of the unit of transfer between the application and the pool, although not necessarily the unit of transfer between the pool and the source file. Applications not knowing the page size of the source file should retrieve the metadata from the file using a page size that is correct for the metadata, then close and reopen the file, or, otherwise determine the page size before calling *mpool_fopen*.

If the *pgin* function is specified, it is called each time a page is read into the memory pool from the source file. If the *pgout* function is specified, it is called each time a page is written to the source file. Both functions are called with the MPOOLFILE pointer returned from *mpool_fopen*, the page number, a pointer to the page being read or written, and the argument *pgcookie*. If either function fails, it should return non-zero and set *errno*, in which case the *db_mpool* function calling it will also fail, leaving *errno* intact.

The *mpool_fclose* function closes the source file indicated by the MPOOLFILE pointer *mpf*. This function does **not** imply a call to *mpool_sync*, i.e. no pages are written to the source file as as a result of calling *mpool_fclose*. The function *mpool_fclose* returns -1 on failure, setting *errno*, and 0 on success.

The function *mpool_get* returns a pointer to the page with the page number specified by *pgnoaddr*, from the source file specified by the MPOOLFILE pointer *mpf*. If the page does not exist or cannot be retrieved, *mpool_get* returns NULL and sets errno.

The *flags* argument is set by *or*'ing any of the following values:

MPOOL_CALLBACK
: After the page number has been determined, but before any other process or thread can access the page, the function specified by the *callback* argument is called. If the function fails, it should return non-zero and set *errno*, in which case *mpool_get* will also fail, leaving *errno* intact. The *callback* function is called with the MPOOLFILE pointer returned from *mpool_fopen* and the page number. This functionality is commonly used when page locking is required, but the page number of the page being retrieved is not known.

MPOOL_CREATE
: If the specified page does not exist, create it.

MPOOL_LAST
: Return the last page of the source file and copy its page number to the location referenced by *pgnoaddr*.

MPOOL_NEW
: Create a new page in the file and copy its page number to the location referenced by *pgnoaddr*.

MPOOL_NOPIN
: Don't pin the page into memory. (This flag is intended for debugging purposes, when it's often useful to examine pages which are currently held by other parts of the application. Pages retrieved in this manner don't need to be returned to the memory pool, i.e. they should **not** be specified as arguments to the *mpool_put* routine.)

Created pages have all their bytes set to 0.

All pages returned by *mpool_get* (unless the MPOOL_NOPIN flag is specified), will be retained (i.e. ''pinned'') in the pool until a subsequent call to *mpool_put*.

The function *mpool_put* indicates that the page referenced by *pgaddr* can be evicted from the pool. *Pgaddr* must be an address previously returned by *mpool_get*.

The flag value is specified by *or*'ing any of the following values:

MPOOL_DIRTY
: The page has been modified and must be written to the source file before being evicted from the pool.

MPOOL_DISCARD
: The page is unlikely to be useful in the near future, and should be discarded before other pages in the pool.

The function *mpool_put* returns -1 on failure, setting *errno*, and 0 on success.

The function *mpool_sync* writes all pages associated with the MPOOLFILE pointer *mpf*, which were specified as arguments to the *mpool_put* function with an associated flag of MPOOL_DIRTY, to the source file. The function *mpool_sync* returns -1 on failure, setting *errno*, and 0 on success.

The function *mpool_unlink* destroys the memory pool identified by the directory *path*, removing all files used to implement the memory pool. (The directory *path* is not removed.) If there are processes which have called *mpool_open* without calling *mpool_close* (i.e., there are processes currently using the memory pool), *mpool_unlink* will fail without further action, unless the force flag is set, in which case *mpool_unlink* will attempt to delete the memory pool files regardless of any processes still using the memory pool. Any accesses to a removed memory pool will likely result in unexpected behavior. The function *mpool_unlink* returns -1 on failure, setting *errno*, and 0 on success.

In the case of catastrophic or system failure, it is possible to clean up a memory pool by removing all of the files in the directory specified to the *mpool_create* function, as memory pool files are never created in any directory other than the one specified to *mpool_create*.

The function *mpool_stat* writes statistics for the memory pool *mp* to the file specified by *fp*. These statistics include the number of files participating in the pool, the active pages in the pool, and numbers as to how effective the cache has been.

ERRORS

The *mpool_create*, *mpool_open* and *mpool_fopen* functions may fail and set *errno* for any of the errors specified for the library functions *open*(2), *read*(2), and *malloc*(3).

The *mpool_close* and *mpool_fclose* functions may fail and set *errno* for any of the errors specified for the library functions *close*(2) and *free*(3).

The *mpool_get* function may fail and set *errno* for any of the errors specified for the library functions *read*(2), *write*(2), and *malloc*(3) or the following:

[EINVAL]
The requested page does not exist and MPOOL_CREATE was not set.

The *mpool_put* function may fail and set *errno* for any of the errors specified for the library function *write*(2) or the following:

[EACCES]
The source file was not opened for writing.

The *mpool_sync* function may fail and set *errno* for any of the errors specified for the library function *write*(2).

The *mpool_unlink* function may fail and set *errno* for any of the errors specified for the library function *unlink*(2) or the following:

[EBUSY]
The memory pool was in use and the force flag was not set.

SEE ALSO

db_btree(3), *db_hash*(3), *db_lock*(3), *db_log*(3), *db_open*(3), *db_recno*(3), *db_txn*(3)

NAME

db_open − database access methods

SYNOPSIS

#include <db.h>

DB *
db_open(const char *file, int flags, int mode,
DBTYPE type, DBINFO *dbinfo, const void *openinfo);

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 transaction, locking, logging and file buffering functionality, as well as various indexed access methods. Many of the functional groups (e.g. the memory pool functions) are useful independently of the rest of the DB functions, although some functional groups are explicitly based on other functional groups (e.g. transactions and logging). For a general description of transactions, see *db_txn*(3). For a general description of the access methods, see *db_open*(3) and then the individual access method manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3) and *db_recno*(3). For a general description of the lock manager, see *db_lock*(3). For a general description of the memory pool manager, see *db_mpool*(3).

This manual page describes the overall structure of the available access methods.

The currently supported file formats are btree, hashed, log and recno (i.e. flat-file oriented). The btree format is a representation of a sorted, balanced tree structure. The hashed format is an extensible, dynamic hashing scheme. The log format is a general-purpose logging facility. The recno format is a byte stream file with fixed or variable length records. The formats and other, format specific information are described in detail in their respective manual pages: *db_btree*(3), *db_hash*(3), *db_log*(3), and *db_recno*(3).

Db_open opens *file* for reading and/or writing. Files never intended to be preserved on disk may be created by setting the file parameter to NULL. (Note, while most of the access methods use *file* as the name of an underlying file on disk, this is not guaranteed. See the manual pages for the individual access methods for more information.)

The *flags* and *mode arguments* are as specified to the *open*(2) function, however, only the O_CREAT, O_EXCL, O_EXLOCK, O_NONBLOCK, O_RDONLY, O_RDWR, O_SHLOCK and O_TRUNC flags are meaningful. (Note, opening a database file O_WRONLY is not possible.)

The *type* argument is of type DBTYPE (as defined in the <db.h> include file) and may be set to DB_BTREE, DB_HASH, DB_LOG or DB_RECNO.

The *dbinfo* argument is a pointer to a structure containing references to locking, logging, transaction, and shared-memory buffer pool information. If *dbinfo* is NULL, then the access method may still use these subsystems, but the usage will be private to the application and managed by DB. If *dbinfo* is non-NULL, then the module referenced by each of the non-NULL fields is used by DB as necessary. The fields of the DBINFO structure are defined as follows:

const char *errpfx;

A prefix to prepend to error messages; used only if *errfile* is non-NULL.

FILE *errfile;

The *stdio*(3) file stream to which error messages are logged.

When any error occurs in the *db_open* function, or in any function called using a field of the returned DB structure, an error value is returned by the function, and the global variable *errno* is set appropriately. In some cases, however, the *errno* value may be insufficient to describe the cause of the error. In these cases, if *errfile* is non-NULL, additional error information will be written to the file stream it represents, preceded by the string, if any, specified by *errpfx*. This error logging facility should not be required for normal operation, but may be useful in debugging applications.

char *errbuf;

The buffer to which error messages are copied. If non-NULL, *errbuf* behaves as described for *errfile*, except that the *errpfx* field is ignored and the error message is copied into the specified buffer instead of being written to the FILE stream. The DB routines assume that the associated buffer is at least 1024 bytes in length.

LOCK_TABLE_T *lockinfo;

If locking is required for the file being opened (as in the case of buffers being maintained in a shared memory buffer pool), the *lockinfo* field contains a return value from the function *lock_open* that should be used (see *db_lock*(3)). If *lockinfo* is NULL, no locking is done.

DB *loginfo;

If modifications to the file being opened should be logged, the *loginfo* field contains a return value from the function *dbopen*, when opening a DB file of type DB_LOG. If *loginfo* is NULL, no logging is done.

MPOOL *mpoolinfo;

If the cache for the file being opened should be maintained in a shared buffer pool, the *mpoolinfo* field contains a return value from the function *mpool_open* that should be used (see *db_mpool*(3)). If *mpoolinfo* is NULL, a memory pool may still be created, but it will be private to the application and managed by DB.

TXNMGR *txninfo;

If the accesses to the file being opened should take place in the context of transactions (providing atomicity and complete error recovery), the *txninfo* 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 *txninfo* is NULL, no transaction support is done.

The *openinfo* argument is a pointer to an access method specific structure described in the access method's manual page. If *openinfo* is NULL, each access method will use defaults appropriate for the system and the access method.

KEY/DATA PAIRS

Access to all access methods is based on key/data pairs. Both keys and data are represented by the following data structure:

```
typedef struct {
        void *data;
        size_t size;
} DBT;
```

The elements of the DBT structure are defined as follows:

data A pointer to a byte string.

size The length of *data*, in bytes.

Key and data byte strings may reference strings of essentially unlimited length, although any two of them must fit into available memory at the same time.

The access methods provide no guarantees about byte string alignment, and applications are responsible for maintaining any necessary alignment.

DB OPERATIONS

Db_open returns a pointer to a DB structure (as defined in the <db.h> include file) on success, and NULL on error. The DB structure 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. Individual access methods specify additional functions and flags which are specific to the method. The fields of the DB structure are as follows:

DBTYPE type;
: The type of the underlying access method (and file format).

int (*close)(const DB *db);
: A pointer to a function to flush any cached information to disk, 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 *close* functions return -1 on failure, setting *errno*, and 0 on success.

int (*del)(const DB *db, TXN *txnid,
    const DBT *key, u_int flags);
: A pointer to a function to remove key/data pairs from the database. The key/data pair associated with the specified *key* are discarded from the database.

  The *txnid* parameter contains a transaction ID returned from *txn_begin*, if the file is being accessed under transaction protection, or NULL if transactions are not in effect.

  The parameter *flag* must be set to 0 or exactly one of the following values:

  R_CURSOR
  : Delete the record referenced by the cursor. The cursor must have previously been initialized.

  The *delete* functions return -1 on error, setting *errno*, 0 on success, and 1 if the specified *key* did not exist in the file.

int (*fd)(const DB *db);
: A pointer to a function which returns a file descriptor representative of the underlying database. A file descriptor referencing the same file will be returned to all processes which call *db_open* with the same *file* name. 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. No file descriptor is available for in memory databases.

  The *fd* functions return -1 on error, setting *errno*, and the file descriptor on success.

int (*get)(const DB *db, TXN *txnid,
    const DBT *key, DBT *data, u_int flags);
: A pointer to a function which is the 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*.

  The *txnid* parameter contains a transaction ID returned from *txn_begin*, if the file is being accessed under transaction protection, or NULL if transactions are not in effect.

  The *get* functions return -1 on error, setting *errno*, 0 on success, and 1 if the *key* was not found.

int (*put)(const DB *db, TXN *txnid,
    DBT *key, const DBT *data, u_int flags);
: A pointer to a function to store key/data pairs in the database.

  The *txnid* parameter contains a transaction ID returned from *txn_begin*, if the file is being accessed under transaction protection, or NULL if transactions are not in effect.

  The parameter *flag* must be set to 0 or exactly one of the following values:

  R_CURSOR
  : Replace the key/data pair referenced by the cursor. The cursor must have previously been initialized.

R_NOOVERWRITE
Enter the new key/data pair only if the key does not previously exist.

The default behavior of the *put* functions is to enter the new key/data pair, replacing any previously existing key.

The *put* functions return -1 on error, setting *errno*, 0 on success, and 1 if the R_NOOVERWRITE *flag* was set and the key already exists in the file.

int (*seq)(const DB *db, TXN *txnid,
DBT *key, DBT *data, u_int flags);

A pointer to a function which is the interface for sequential retrieval from the database. The address and length of the key are returned in the structure referenced by *key*, and the address and length of the data are returned in the structure referenced by *data*.

The *txnid* parameter contains a transaction ID returned from *txn_begin*, if the file is being accessed under transaction protection, or NULL if transactions are not in effect.

Sequential key/data pair retrieval may begin at any time, and the logical position of the "cursor" is not affected by calls to the *del*, *get*, *put*, or *sync* functions. Modifications to the database during a sequential scan will be reflected in the scan, i.e. records inserted behind the cursor will not be returned while records inserted in front of the cursor will be returned.

The parameter *flag* must be set to 0 or exactly one of the following values:

R_CURSOR
The data associated with the specified key is returned. This differs from the *get* functions in that it sets or initializes the cursor to the location of the key as well.

R_FIRST
The first key/data pair of the database is returned, and the cursor is set or initialized to reference it.

R_NEXT
Retrieve the key/data pair immediately after the cursor. If the cursor is not yet set, this is the same as the R_FIRST flag.

The *seq* functions return -1 on error, setting *errno*, 0 on success, and 1 if there are no key/data pairs less than or greater than the specified or current key.

int (*sync)(const DB *db, u_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 parameter *flag* must be set to 0 or a value specified by an access method specific manual page.

The *sync* functions return -1 on failure, setting *errno*, and 0 on success.

ERRORS

The *db_open* function may fail and set *errno* for any of the errors specified for the library functions *open*(2), *malloc*(3) or the following:

[EFTYPE]
A file is incorrectly formatted.

[EINVAL]
A parameter has been specified (hash function, recno pad byte etc.) that is incompatible with the current file specification or, a flag to a function which is not meaningful for the function (for example, use of the cursor without prior initialization) or there is a mismatch between the version number of file and the software.

The *close* functions may fail and set *errno* for any of the errors specified for the library functions *close*(2), *read*(2), *write*(2), *free*(3), or *fsync*(2).

The *del*, *get*, *put* and *seq* functions may fail and set *errno* for any of the errors specified for the library functions *read*(2), *write*(2), *free*(3) or *malloc*(3).

The *fd* functions will fail and set *errno* to ENOENT for in memory databases.

The *sync* functions may fail and set *errno* for any of the errors specified for the library function *fsync*(2).

SEE ALSO

db_btree(3), *db_hash*(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_recno*(3), *db_txn*(3)

BUGS

The name DBT is a mnemonic for ‘‘data base thang’’, and was used because noone could think of a reasonable name that wasn't already in use somewhere else.

The *fd* function interface is a kluge, and will be deleted in a future version of the interface.

Only big and little endian byte order is supported.

NAME

db_recno – record number database access method

DESCRIPTION

specific details of the recno access method.

ACCESS METHOD SPECIFIC INFORMATION

The recno access method specific data structure provided to *db_open* is typedef'd and named RECNOINFO. A RECNOINFO structure has at least the following fields, which may be initialized before calling *db_open*:

u_int8_t bval;

The delimiting byte to be used to mark the end of a record for variable-length records, and the pad character for fixed-length records. If no value is specified, newlines (''\n'') are used to mark the end of variable-length records and fixed-length records are padded with spaces.

char *bfname;

The recno access method stores the in-memory copies of its records in a btree. If bfname is non-NULL, it specifies the name of the btree file, as if specified as the file name for a *db_open* of a btree file.

u_int cachesize;

A suggested maximum size, in bytes, of the memory cache. This value is **only** advisory, and the access method will allocate more memory rather than fail. If *cachesize* is 0 (no size is specified) a default size is used.

u_long flags;

The flag value is specified by *or*'ing any of the following values:

R_FIXEDLEN

The records are fixed-length, not byte delimited. The structure element *reclen* specifies the length of the record, and the structure element *bval* is used as the pad character. Any records, inserted into the database, that are less than *reclen* bytes long are automatically padded.

R_NOKEY

In the interface specified by *db_open*, the sequential record retrieval fills in both the caller's key and data structures. If the R_NOKEY flag is specified, the *cursor* functions are not required to fill in the key structure. This permits applications to retrieve records at the end of files without reading all of the intervening records.

R_SNAPSHOT

This flag requires that a snapshot of the file be taken when *db_open* is called, instead of permitting any unmodified records to be read from the original file.

int 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 would be the number 4,321. If *lorder* is 0 (no order is specified) the current host order is used.

u_int psize;

The recno access method stores the in-memory copies of its records in a btree. This value is the size (in bytes) of the pages used for nodes in that tree. If *psize* is 0 (no page size is specified) a page size is chosen based on the underlying file system I/O block size. See *btree*(3) for more information.

size_t reclen;

The length of a fixed-length record.

DB OPERATIONS

The data part of the key/data pair used by the recno access method is the same as other access methods. The key is different. The *data* field of the key should be a pointer to a memory location of type *recno_t*, as defined in the <db.h> include file. This type is normally the largest unsigned integral type available to the implementation. The *size* field of the key should be the size of that type.

The record number data structure is either variable or fixed-length records stored in a flat-file format, accessed by the logical record number. The existence of record number five requires the existence of records one through four, and the deletion of record number one causes record number five to be renumbered to record number four, as well as the cursor, if positioned after record number one, to shift down one record. The creation of record number five when records one through four do not exist causes the logical creation of them with zero-length data.

Because there is no meta-data associated with the underlying recno access method files, any changes made to the default values (e.g. fixed record length or byte separator value) must be explicitly specified each time the file is opened.

The functions returned by *db_open* for the btree access method are as described in *db_open*(3), with the following exceptions and additions:

type The type is DB_RECNO.

put Using the *put* interface to create a new record will cause the creation of multiple, empty records if the record number is more than one greater than the largest record currently in the database.

The *put* function takes the following additional flags:

R_IAFTER
Append the data immediately after the data referenced by *key*, creating a new key/data pair. The record number of the appended key/data pair is returned in the *key* structure.

R_IBEFORE
Insert the data immediately before the data referenced by *key*, creating a new key/data pair. The record number of the inserted key/data pair is returned in the *key* structure.

R_SETCURSOR
Store the key/data pair, setting or initializing the position of the cursor to reference it.

seq The *seq* function takes the following additional flags:

R_LAST
The last key/data pair of the database is returned, and the cursor is set or initialized to reference it.

R_PREV
Retrieve the key/data pair immediately before the cursor. If the cursor is not yet set, this is the same as the R_LAST flag.

If the database file is a character special file and no complete key/data pairs are currently available, the *seq* function returns 2.

sync The *sync* function takes the following additional flag:

R_RECNOSYNC
This flag causes the *sync* function to apply to the btree file which underlies the recno file, not the recno file itself. (See the *bfname* field of RECNOINFO structure, above, for more information.)

ERRORS

The *recno* access method functions may fail and set *errno* for any of the errors specified for the library function *db_open*(3) or the following:

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

SEE ALSO

db_btree(3), *db_hash*(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_txn*(3)

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.

BUGS

The *sync* function's R_RECNOSYNC interface is a kluge, and will be deleted in a future version of the interface.

NAME

db_txn − transaction management functions

SYNOPSIS

#include <db.h>
#include <db_lock.h>

int
txn_create(const char *path, mode_t mode, u_int maxtxns, u_int flags);

TXNMGR *
txn_open(const char *path, DBT *logp, LOCK_TABLE_T *lockp,
int (*recover)(DBT *lsn, DBT *log_entry, int isundo));

TXN *
txn_begin(TXNMGR *txnp);

int
txn_commit(TXN *tid);

int
txn_prepare(TXN *tid);

int
txn_abort(TXN *tid);

int
txn_close(TXNMGR *txnp);

int
txn_unlink(const char *path, int force);

DESCRIPTION

specific details of the transaction support.

Db_txn is the library interface that provides transaction semantics. Full transaction support is provided by a collection of modules that provide well defined interfaces to the services required for transaction processing. These services are recovery (see *db_log*(3)), concurrency control (see *db_lock*(3)), and the management of shared data (see *db_mpool*(3)). Transaction semantics can be applied to the access methods described in *db*(3) through function call parameters.

The model intended for transactional use (and that is used by the access methods) is that write-ahead logging is provided by *db_log*(3) to record both before- and after-image logging. Locking follows a two-phase protocol and is implemented by *db_lock*(3).

The function *txn_create* creates and initializes the transaction region identified by the *path* directory. This directory must already exist when *txn_create* is called. If the transaction region identified by *path* already exists, then *txn_create* returns success without further action. The files associated with the transaction region are created in the directory specified by *path*. (The group of the created files is based on the system and directory defaults, and is not further specified by *txn_create*.) All files created by *txn_create* are created with mode *mode* (as described in *chmod*(2)) and modified by the process' umask value (see *umask*(2)). Any necessary, associated log and lock regions are created as well (see *db_log*(3) and *db_lock*(3)).

The *maxtxns* argument specifies the maximum number of simultaneous transactions that are supported. This bounds the size of backing files and is used to derive limits for the size of the lock region and logfiles.

When there are more than *maxtxns* concurrent transactions, calls to *txn_begin* may fail.

Default locking and logging protocols are provided only if the backing files exist. If the backing files do not exist, the *flags* parameter must indicate both a logging mode and locking mode specified by *or*'ing together at most one flag from each of the TXN_LOCK and TXN_LOG classes as follows:

TXN_LOCK_2PL
: Use two-phase locking.

TXN_LOCK_OPTIMISTIC
: Use optimistic locking (not currently implemented).

TXN_LOG_REDO
: Provide redo-only logging (not currently implemented).

TXN_LOG_UNDO
: Provide undo-only logging (not currently implemented).

TXN_LOG_UNDOREDO
: Provide undo/redo write-ahead logging.

The function *txn_create* returns -1 on failure, setting *errno*, and 0 on success.

The function *txn_open* returns a pointer to the transaction region identified by *path*, which must have already been created by a call to *txn_create*. The process must have permission to read and write files with owners, groups and permissions as described for *txn_create*. The *txn_open* function returns NULL on failure, setting *errno*.

The *recover* argument specifies a function that is called by *txn_abort* during transaction abort. This function takes three arguments:

lsn
: A log sequence number (LSN).

log_entry
: A log record.

isundo
: An undo flag set to 0 if the operation is a redo and set to 1 if the operation an undo.

As discussed in the *db_log (3)* manual page, the application is responsible for providing any necessary structure to the log record. For example, the application must understand what part of the log record is an operation code, what part is redo information, and what part is undo information.

The *txn_begin* function creates a new transaction in the designated transaction manager, returning a pointer to a TXN that uniquely identifies it.

The *txn_commit* function ends the transaction specified by the *tid* argument. Any locks held by the transaction are released. If logging is enabled, a commit log record is written and flushed to disk.

The *txn_abort* function causes an abnormal termination of the transaction. If logging is enabled, the log is played backwards and any recovery operations are initiated through the *recover* function specified to *txn_open*. After recovery is completed, all locks held by the transaction are released.

The *txn_close* function detaches a process from the transaction environment specified by the TXNMGR pointer. All mapped regions are unmapped and any allocated resources are freed. Any uncommitted transactions are aborted.

The function *txn_unlink* destroys the transaction region identified by the directory *path*, removing all files used to implement the transaction region. (The directory *path* is not removed.) If there are processes which have called *txn_open* without calling *txn_close* (i.e., there are processes currently using the transaction region), *txn_unlink* will fail without further action, unless the force flag is set, in which case *txn_unlink* will attempt to delete the transaction region files regardless of any processes still using the transaction region. Any accesses to a removed transaction region will likely result in unexpected behavior. The function *txn_unlink* returns -1 on failure, setting *errno*, and 0 on success.

In the case of catastrophic or system failure, it is possible to clean up a transaction region by removing all of the files in the directory specified to the *txn_create* function, as transaction region files are never created in any directory other than the one specified to *txn_create*.

The *txn_prepare* function initiates the beginning of a two phase commit. In a distributed transaction, the prepare directive should be issued to all participating transaction managers. Each manager must take whatever action is necessary to guarantee that a future call to *txn_commit* on the specified *tid* will succeed.

SYSTEM INTEGRATION

This model can be applied to data bases other than the provided access methods. For example, consider an application that provides transaction semantics to data stored in regular files accessed using the *read*(2) and *write*(2) system calls. The operations for which transaction protection is desired are bracketed by calls to *txn_begin* and *txn_commit*.

Before data are referenced, a call is made to the lock manager, *db_lock*, for a lock of the appropriate type (e.g. read) on the object being locked. The object might be a page in the file, a byte, a range of bytes, or some key. Before a write is performed, the application makes a call to the log manager, *db_log*, to record enough information to redo the operation in case of failure after commit and to undo the operation in case of abort. After the log message is written, the write system calls are issued. After all requests are issued, the application calls *txn_commit*. When *txn_commit* returns, the caller is guaranteed that all necessary log writes have been written to disk.

At any time, the application may call *txn_abort*, which will result in the appropriate calls to the *recover* routine to restore the ‘‘database’’ to a consistent pre-transaction state. (The recover routine must be able to either reapply or undo the update depending on the context, for each different type of log record.)

If the application should crash, the recovery process uses the *db_log* interface to read the log and call the *recover* routine to restore the database to a consistent state.

The *txn_prepare* function provides the core functionality to implement distributed transactions, but it does not actually manage the notification of distributed transaction managers. The caller is responsible for issuing *txn_prepare* calls to all sites participating in the transaction. If all responses are positive, the caller can issue a *txn_commit*. If any of the responses are negative, the caller should issue a *txn_abort*. In general, the *txn_prepare* call requires that the transaction log be flushed to disk.

The structure of the transaction support allows application designers to trade off performance and protection. Since DB manages many structures in shared memory, its information is subject to corruption by applications when the library is linked directly with the application. For this reason, DB is designed to allow compilation into a separate server process that may be accessed via a socket interface. In this way DB's data structures are protected from application code, but communication overhead is increased. When applications are trusted, DB may be compiled directly into the application for increased performance.

ERRORS

The *txn_create* function may fail and set *errno* for any of the errors specified for the library functions *open*(2), *write*(2), *malloc*(3), *lock_create*(3), and *log_create*(3).

The *txn_open* function may fail and set *errno* to any of the errors specified for the library functions *open*(2), *write*(2), *malloc*(3), *lock_open*(3), and *log_open*(3).

The *txn_begin* function may fail and set *errno* to ENOSPC indicating that the maximum number of concurrent transactions has been reached.

The *txn_commit* function may fail and set *errno* to EINVAL indicating that the transaction was aborted.

The *txn_close* function may fail and set *errno* to any of the errors specified for the library functions *close*(2), *read*(2), *write*(2), *free*(3), *fsync*(2), *lock_close*(3) or *log_close*(3).

The *txn_unlink* function may fail and set *errno* to any of the errors specified for the library functions *unlink*(2), *lock_unlink*(3), and *log_unlink*(3), or the following:

[EBUSY]
The transaction region was in use and the force flag was not set.

SEE ALSO

db_btree(3), *db_hash*(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_recno*(3)

LIBTP: Portable, Modular Transactions for UNIX, Margo Seltzer, Michael Olson, USENIX proceedings, Winter 1992.

BUGS

The *maxtxns* parameter is a kluge, and should be deleted in favor of dynamically expanding the transaction region.