NAME

Db − database access class

SYNOPSIS

```
#include <db_cxx.h>

static int
Db::open(const char *fname, DBTYPE type,
        int flags, int mode, DbEnv *dbenv, DbInfo *dbinfo, Db **dbpp);

DBTYPE
Db::get_type(void) const;

int
Db::close(int flags);

int
Db::cursor(DbTxn *txnid, Dbc **cursorp);

int
Db::del(Dbt *key, DbTxn *txnid);

int
Db::fd(int *fdp);

int
Db::get(DbTxn *txnid, Dbt *key, Dbt *data, int flags);

int
Db::put(DbTxn *txnid, Dbt *key, Dbt *data, int flags);

int
Db::stat(void *sp, void *(*db_malloc)(size_t), int flags);

int
Db::sync(int flags);
```

DESCRIPTION

The DB library is a family of classes 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 classes (e.g., the file page caching class) are useful independent of the other DB classes, although some classes are explicitly based on other classes (e.g., transactions and logging). For a general description of the DB package, see *db_intro*(3).

This manual page describes the Db class, which is the center of access activity.

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, using the Dbt class. See *Dbt*(3) for specific information on the structure and capabilities of a Dbt.

The *Db::open* method 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* method copies a pointer to a Db object into the memory location referenced by *dbpp*. The methods of this object allow you to perform various database actions, as described below. The *Db::open* method throws a *DbException*(3) or 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_cxx.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:

DB_CREATE
: 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 *DbMpool*(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.

DB_THREAD
: Cause the Db handle returned by the *Db::open* method 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.

See *DbEnv*(3) for a description of the *dbenv* argument, and *DbInfo*(3) for a description of the *dbinfo* argument.

Db OPERATIONS

The Db object returned by *Db::open* describes a database type, and includes a set of functions to perform various actions, as described below. The methods for Db are as follows:

DBTYPE Db::get_type(void);
: The type of the underlying access method (and file format). Returns one of DB_BTREE, DB_HASH or DB_RECNO. This value 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 Db::close(int flags);
: A method to flush any cached information to disk, close any open cursors (see *Dbc*(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* method 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 method.

The *close* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

int Db::cursor(DbTxn *txnid, Dbc **cursorp);

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

A cursor is an object used to provide sequential access through a database.

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 *Dbc*(3) for more information.

The *cursor* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

int Db::del(DbTxn *txnid, Dbt *key, int flags);

A method 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* method throws a *DbException*(3) or returns the value of *errno* on failure, 0 on success, and DB_NOTFOUND if the specified *key* did not exist in the file.

int Db::fd(int *fdp);

A method 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* method only supports a coarse-grained form of locking. Applications should use the lock manager where possible.

The *fd* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

int Db::get(DbTxn *txnid, Dbt *key, Dbt *data, int flags);

A method 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 object 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 *Dbc*(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_GET_RECNO
: Retrieve a specific numbered record 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* method.

  For DB_GET_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)). In this case, the *data* field of the *key* must be a pointer to a memory location of type *db_recno_t*, as described in *Dbt*(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* method returns DB_KEYEMPTY. Otherwise, if the requested key isn't in the database, the *get* method returns DB_NOTFOUND. Otherwise, the *get* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

int Db::put(DbTxn *txnid, Dbt *key, Dbt *data, int flags);

A method to store key/data pairs in the database. If the database supports duplicates, the *put* method 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* method 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* method throws a *DbException*(3) or 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 Db::sync(int flags);

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

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

See the *close* method description above for a discussion of Db and cached data.

The *sync* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

int Db::stat(void *sp,
        void *(*db_malloc)(size_t), int flags);

A method 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 method 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
: In the case of a btree or recno database, fill in the *bt_nrecs* field, 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.

The *stat* method throws a *DbException*(3) or 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_cxx.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 file *examples_cxx/AccessExample.cpp* in the Db source distribution for a C++ language code example of how such an application might use the Db library.

ERRORS

Methods marked as returning *errno* will, by default, throw an exception that encapsulates the error information. The default error behavior can be changed, see *DbException*(3).

The *Db::open* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: Db::sync(3), DbLock::get(3), DbLock::put(3), DbLockTab::id(3), DbLockTab::vec(3), DbLog::db_register(3), DbLog::put(3), DbMpool::close(3), DbMpool::db_register(3), DbMpool::open(3), DbMpoolFile::close(3), DbMpoolFile::get(3), DbMpoolFile::open(3), DbMpoolFile::put(3), DbMpoolFile::set(3), DbMpoolFile::sync(3), calloc(3), close(2), fcntl(2), fflush(3), malloc(3), memcpy(3), memmove(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* method may fail and throw a *DbException*(3) or 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 method, 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 DbEnv 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* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: Db::sync(3), DbLock::get(3), DbLock::put(3), DbLockTab::vec(3), DbLog::db_register(3), DbLog::put(3), DbMpool::close(3), DbMpoolFile::close(3), DbMpoolFile::get(3), DbMpoolFile::put(3), DbMpoolFile::set(3), DbMpoolFile::sync(3), calloc(3), close(2), fflush(3), malloc(3), memcpy(3), memmove(3), memset(3), munmap(2), realloc(3), and strerror(3).

The *Db::cursor* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: calloc(3).

In addition, the *Db::cursor* method may fail and throw a *DbException*(3) or 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* method may fail and throw a *DbException*(3) or return *errno* for any of the errors specified for the following DB and library functions: DbLock::get(3), DbLock::put(3), DbLockTab::id(3), DbLockTab::vec(3), DbLog::put(3), DbMpoolFile::get(3), DbMpoolFile::put(3), DbMpoolFile::set(3), calloc(3), fcntl(2), fflush(3), malloc(3), memcmp(3), memcpy(3), memmove(3), memset(3), realloc(3), and strerror(3).

In addition, the *Db::del* method may fail and throw a *DbException*(3) or 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* method may fail and throw a *DbException*(3) or return *errno* for the following conditions:

[ENOENT]
The *Db::fd* method 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* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLock::get(3), DbLock::put(3), DbLockTab::id(3), DbLockTab::vec(3), DbLog::put(3), DbMpoolFile::get(3), DbMpoolFile::put(3), DbMpoolFile::set(3), Dbc::get(3), calloc(3), fcntl(2), fflush(3), malloc(3), memcmp(3), memcpy(3), memmove(3), memset(3), realloc(3), and strerror(3).

In addition, the *Db::get* method may fail and throw a *DbException*(3) or 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) method 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* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLock::get(3), DbLock::put(3), DbLockTab::id(3), DbLockTab::vec(3), DbLog::put(3), DbMpoolFile::get(3), DbMpoolFile::put(3), DbMpoolFile::set(3), calloc(3), fcntl(2), fflush(3), malloc(3), memcmp(3), memcpy(3), memmove(3), memset(3), realloc(3), and strerror(3).

In addition, the *Db::put* method may fail and throw a *DbException*(3) or 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* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLock::get(3), DbLock::put(3), DbLockTab::id(3), DbLockTab::vec(3), DbMpoolFile::get(3), DbMpoolFile::put(3), calloc(3), fcntl(2), fflush(3), malloc(3), memcpy(3), and memset(3).

The *Db::sync* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: Db::get(3), Db::sync(3), DbLock::get(3), DbLock::put(3), DbLockTab::id(3), DbLockTab::vec(3), DbLog::put(3), DbMpoolFile::get(3), DbMpoolFile::put(3), DbMpoolFile::set(3), DbMpoolFile::sync(3), calloc(3), close(2), fcntl(2), fflush(3), malloc(3), memcpy(3), memmove(3), memset(3), munmap(2), open(2), realloc(3), strerror(3), unlink(2), and write(2).

In addition, the *Db::sync* method may fail and throw a *DbException*(3) or 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_internal*(3), *db_thread*(3), *Db*(3), *Dbc*(3), *DbEnv*(3), *DbException*(3), *DbInfo*(3), *DbLock*(3), *DbLockTab*(3), *DbLog*(3), *DbLsn*(3), *DbMpool*(3), *DbMpoolFile*(3), *Dbt*(3), *DbTxn*(3), *DbTxnMgr*(3)