NAME

DbTxnMgr – Db transaction management

SYNOPSIS

#include <db_cxx.h>

static int
DbTxnMgr::open(const char *dir,
u_int32_t flags, int mode, DbEnv *dbenv, DbTxnMgr **regionp);

int
DbTxnMgr::begin(DbTxn *pid, DbTxn **tid);

int
DbTxnMgr::checkpoint(u_int32_t kbyte, u_int32_t min) const;

int
DbTxnMgr::close();

static int
DbTxnMgr::unlink(const char *dir, int force, DbEnv *dbenv);

int
DbTxnMgr::stat(DB_TXN_STAT **statp, void *(*db_malloc)(size_t));

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 specific details of the Db transaction support. The DbTxnMgr class is used in conjunction with *DbTxn*(3) to provide transaction semantics. Full transaction support is provided by a collection of modules that provide interfaces to the services required for transaction processing. These services are recovery (see *DbLog*(3)), concurrency control (see *DbLock*(3) and *DbLockTab*(3)), and the management of shared data (see *DbMpool*(3) and *DbMpoolFile*(3)). Transaction semantics can be applied to the access methods described in *Db*(3) through method call parameters.

The model intended for transactional use (and that is used by the access methods) is that write-ahead logging is provided by *DbLog*(3) to record both before- and after-image logging. Locking follows a two-phase protocol (i.e., all locks are released at transaction commit).

DbTxnMgr::open

The *DbTxnMgr::open* method copies a pointer, to the transaction region identified by the **directory** *dir*, into the memory location referenced by *regionp*.

If the *dbenv* argument to *DbTxnMgr::open* was initialized using *DbEnv::appinit*, *dir* is interpreted as described by *DbEnv*(3).

Otherwise, if *dir* is not NULL, it is interpreted relative to the current working directory of the process. If *dir* is NULL, the following environment variables are checked in order: ‘‘TMPDIR’’, ‘‘TEMP’’, and ‘‘TMP’’. If one of them is set, transaction region files are created relative to the directory it specifies. If none of them are set, the first possible one of the following directories is used: */var/tmp*, */usr/tmp*, */temp*, */tmp*, *C:/temp* and *C:/tmp*.

All files associated with the transaction region are created in this directory. This directory must already exist when DbTxnMgr::open is called. If the transaction region already exists, the process must have permission to read and write the existing files. If the transaction region does not already exist, it is optionally created and initialized.

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_THREAD
Cause the DbTxnMgr handle returned by the *DbTxnMgr::open* method to be useable by multiple threads within a single address space, i.e., to be ''free-threaded''.

DB_TXN_NOSYNC
On transaction commit, do not synchronously flush the log. This means that transactions exhibit the ACI (atomicity, consistency and isolation) properties, but not D (durability), i.e., database integrity will be maintained but it is possible that some number of the most recently committed transactions may be undone during recovery instead of being redone.

The number of transactions that are potentially at risk is governed by how often the log is checkpointed (see *db_checkpoint*(1)) and how many log updates can fit on a single log page.

All files created by the transaction subsystem 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.

The transaction subsystem is configured based on which set methods have been used. It is expected that applications will use a single DbEnv object as the argument to all of the subsystems in the DB package. The fields of the DbEnv object used by *DbTxnMgr::open* are described below. As references to the DbEnv object may be maintained by *DbTxnMgr::open*, it is necessary that the DbEnv object and memory it references be valid until the object is destroyed. Any of the DbEnv fields that are not explicitly set will default to appropriate values.

The following fields in the DbEnv object may be initialized, using the appropriate set method, before calling *DbTxnMgr::open*:

void *(*db_errcall)(char *db_errpfx, char *buffer);
FILE *db_errfile;
const char *db_errpfx;
class ostream *db_error_stream;
int db_verbose;
The error fields of the DbEnv behave as described for *DbEnv*(3).

DbLog *lg_info;
The logging region that is being used for this transaction environment. The *lg_info* field contains a return value from the method *DbLog::open*. **Logging is required for transaction environments, and it is an error to not specify a logging region.**

DbLockTab *lk_info;
The locking region that is being used for this transaction environment. The *lk_info* field contains a return value from the method *DbLockTab::open*. If *lk_info* is NULL, no locking is done in this transaction environment.

u_int32_t tx_max;
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 *tx_max* concurrent transactions, calls to *DbTxnMgr::begin* may cause backing files to grow. If *tx_max* is 0, a default value is used.

int DbTxnMgr::recover(DbLog *logp, Dbt *DbLog::rec,
DbLsn *lsnp, int redo, void *info);
A method that is called by *DbTxn::abort* during transaction abort. This method takes five arguments:

logp A pointer to the transaction log (DbLog *).

DbLog::rec
A log record.

lsnp A pointer to a log sequence number (DbLsn *).

redo An integer value that is set to one of the following values:

DB_TXN_BACKWARD_ROLL
The log is being read backward to determine which transactions have been committed and which transactions were not (and should therefore be aborted during recovery).

DB_TXN_FORWARD_ROLL
The log is being played forward, any transaction ids encountered that have not been entered into the list referenced by *info* should be ignored.

DB_TXN_OPENFILES
The log is being read to open all the files required to perform recovery.

DB_TXN_REDO
Redo the operation described by the log record.

DB_TXN_UNDO
Undo the operation described by the log record.

info An opaque pointer used to reference the list of transaction IDs encountered during recovery.

If *recover* is NULL, the default is that only Db access method operations are transaction protected, and the default recover method will be used.

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

DbTxnMgr::begin

The *DbTxnMgr::begin* method creates a new transaction in the transaction manager, copying a pointer to a DbTxn that uniquely identifies it into the memory referenced by *tid*. If the *pid* argument is non-NULL, the new transaction is a nested transaction with the transaction indicated by *pid* as its parent.

Transactions may not span threads, i.e., each transaction must begin and end in the same thread, and each transaction may only be used by a single thread.

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

DbTxnMgr::close

The *DbTxnMgr::close* method detaches a process from the transaction environment specified by the DbTxnMgr object. All mapped regions are unmapped and any allocated resources are freed. Any uncommitted transactions are aborted.

In addition, if the *dir* argument to *DbTxnMgr::open* was NULL and *dbenv* was not initialized using *DbEnv::appinit*, all files created for this shared region will be removed, as if *DbTxnMgr::unlink* were called.

When multiple threads are using the DbTxnMgr handle concurrently, only a single thread may call the *DbTxnMgr::close* method.

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

DbTxnMgr::unlink

The *DbTxnMgr::unlink* method destroys the transaction region identified by the directory *dir*, removing all files used to implement the transaction region. (The directory *dir* is not removed.) If there are processes that have called *DbTxnMgr::open* without calling *DbTxnMgr::close* (i.e., there are processes currently using the transaction region), *DbTxnMgr::unlink* will fail without further action, unless the force flag is set, in which case *DbTxnMgr::unlink* will attempt to remove the transaction region files regardless of any processes still using the transaction region.

The result of attempting to forcibly destroy the region when a process has the region open is unspecified. Processes using a shared memory region maintain an open file descriptor for it. On UNIX systems, the region removal should succeed and processes that have already joined the region should continue to run in the region without change, however processes attempting to join the transaction region will either fail or attempt to create a new region. On other systems, e.g., WNT, where the *unlink*(2) system call will fail if any process has an open file descriptor for the file, the region removal will fail.

In the case of catastrophic or system failure, database recovery must be performed (see *db_recover*(1) or the DB_RECOVER and DB_RECOVER_FATAL flags to *DbEnv::appinit*(3)). Alternatively, if recovery is not required because no database state is maintained across failures, it is possible to clean up a transaction region by removing all of the files in the directory specified to the *DbTxnMgr::open* method, as transaction region files are never created in any directory other than the one specified to *DbTxnMgr::open*. Note, however, that this has the potential to remove files created by the other DB subsystems in this database environment.

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

DbTxnMgr::checkpoint

The *DbTxnMgr::checkpoint* method syncs the underlying memory pool, writes a checkpoint record to the log and then flushes the log.

If either *kbyte* or *min* is non-zero, the checkpoint is only done if more than *min* minutes have passed since the last checkpoint, or if more than *kbyte* kilobytes of log data have been written since the last checkpoint.

The *DbTxnMgr::checkpoint* method throws a *DbException*(3) or returns the value of *errno* on failure, 0 on success, and DB_INCOMPLETE if there were pages that needed to be written but that *DbMpool::sync (3)* (see *DbMpool (3))* was unable to write immediately. In this case, the *DbTxnMgr::checkpoint* call should be retried.

The *DbTxnMgr::checkpoint* method is based on the C *txn_checkpoint* function, which is the underlying function used by the *db_checkpoint*(1) utility. See the source code for the *db_checkpoint* utility for an example of using *txn_checkpoint* in a UNIX environment.

DbTxnMgr::stat

The *DbTxnMgr::stat*

method creates a statistical structure and copies pointers to it into user-specified memory locations.

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.

The transaction region statistics are stored in a structure of type DB_TXN_STAT (typedef'd in <db_cxx.h>). The following DB_TXN_STAT fields will be filled in:

DbLsn st_last_ckp;

The LSN of the last checkpoint.

DbLsn st_pending_ckp;
The LSN of any checkpoint that is currently in progress. If *st_pending_ckp* is the same as *st_last_ckp* there is no checkpoint in progress.
time_t st_time_ckp;
The time the last completed checkpoint finished (as returned by *time*(2)).
u_int32_t st_last_txnid;
The last transaction ID allocated.
u_int32_t st_maxtxns;
The maximum number of active transactions supported by the region.
u_int32_t st_naborts;
The number of transactions that have aborted.
u_int32_t st_nactive;
The number of transactions that are currently active.
u_int32_t st_nbegins;
The number of transactions that have begun.
u_int32_t st_ncommits;
The number of transactions that have committed.
u_int32_t ;
The number of times that a thread of control was forced to wait before obtaining the region lock.
u_int32_t ;
The number of times that a thread of control was able to obtain the region lock without waiting.

DB_TXN_ACTIVE *st_txnarray;
A pointer to an array of *st_nactive* DB_TXN_ACTIVE structures, describing the currently active transactions. The following fields of the DB_TXN_ACTIVE structure (typedef'd in <db_cxx.h>) will be filled in:

u_int32_t txnid;
The transaction ID as returned by *DbTxnMgr::begin*(3).
DbLsn lsn;
The LSN of the transaction-begin record.

ENVIRONMENT VARIABLES

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

DB_HOME
If the *dbenv* argument to *DbTxnMgr::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 *DbTxnMgr::open*, as described in *db_appinit*(3).

TMPDIR
If the *dbenv* argument to *DbTxnMgr::open* was NULL or not initialized using *db_appinit*, the environment variable TMPDIR may be used as the directory in which to create the transaction region, as described in the *DbTxnMgr::open* section above.

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 *DbTxnMgr::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: DbTxnMgr::unlink(3), close(2), db_version(3), fcntl(2), fflush(3), lseek(2), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), open(2), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), time(3), unlink(2), and write(2).

In addition, the *DbTxnMgr::open* method may fail and throw a *DbException*(3) or return *errno* for the

following conditions:

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

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

The *dbenv* parameter was NULL.

[EAGAIN]
The shared memory region was locked and (repeatedly) unavailable.

The *DbTxnMgr::begin* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLog::put(3), fcntl(2), fflush(3), lseek(2), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), strerror(3), and write(2).

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

[ENOSPC]
The maximum number of concurrent transactions has been reached.

The *DbTxnMgr::checkpoint* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLog::compare(3), DbLog::put(3), DbMpool::sync(3), fcntl(2), fflush(3), malloc(3), memcpy(3), memset(3), strerror(3), and time(3).

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

The *DbTxnMgr::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: DbLog::flush(3), DbTxn::abort(3), close(2), fcntl(2), fflush(3), munmap(2), and strerror(3).

The *DbTxnMgr::unlink* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: close(2), fcntl(2), fflush(3), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), open(2), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), and unlink(2).

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

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

The *DbTxnMgr::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: fcntl(2), and malloc(3).

SEE ALSO

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

BUGS

Nested transactions are not yet implemented.

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)