NAME

DbTxn − Db transaction management

SYNOPSIS

#include <db_cxx.h>

int
DbTxn::prepare();

int
DbTxn::commit();

int
DbTxn::abort();

u_int32_t
DbTxn::id();

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 DbTxn class is used in conjunction with *DbTxnMgr*(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).

DbTxn::prepare

The *DbTxn::prepare* method initiates the beginning of a two phase commit. In a distributed transaction environment, *db* can be used as a local transaction manager. In this case, the distributed transaction manager must send *prepare* messages to each local manager. The local manager must then issue a *DbTxn::prepare* and await its successful return before responding to the distributed transaction manager. Only after the distributed transaction manager receives successful responses from all of its *prepare* messages should it issue any *commit* messages.

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

DbTxn::commit

The *DbTxn::commit* method ends the transaction associated with the DbTxn. If DB_TXN_NOSYNC was not specified, a commit log record is written and flushed to disk, as are all previously written log records. If the transaction is nested, its locks are acquired by the parent transaction, otherwise its locks are released. Any applications that require strict two-phase locking must not release any locks explicitly, leaving them all to be released by *DbTxn::commit*.

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

DbTxn::abort

The *DbTxn::abort* method causes an abnormal termination of the transaction. The log is played backwards and any necessary recovery operations are initiated through the *recover* method specified to *DbTxnMgr::open*. After recovery is completed, all locks held by the transaction are acquired by the parent transaction in the case of a nested transaction or released in the case of a non-nested transaction. As is the case for *DbTxn::commit*, applications that require strict two phase locking should not explicitly release any locks.

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

DbTxn::id

The *DbTxn::id* method returns the unique transaction id associated with the specified transaction. Locking calls made on behalf of this transaction should use the value returned from *DbTxn::id* as the locker parameter to the *DbLockTab::get* or *DbLockTab::vec* calls.

TRANSACTIONS

Creating transaction protected applications using the Db access methods requires little system customization. In most cases, the default parameters to the locking, logging, memory pool, and transaction subsystems will suffice. Applications can use *DbEnv::appinit* (see *DbEnv*(3)) to perform this initialization, or they may do it explicitly.

Each database operation (i.e., any call to a method underlying the handles returned by *Db::open* and *Db::cursor* described in *Db*(3)) is normally performed on behalf of a unique locker. If multiple calls on behalf of the same locker are desired, then transactions must be used.

Once the application has initialized the Db subsystems that it is using, it may open the Db access method databases. For applications performing transactions, the databases must be opened after subsystem initialization, and cannot be opened as part of a transaction. Once the databases are opened, the application can group sets of operations into transactions, by surrounding the operations with the appropriate *DbTxnMgr::begin*, *DbTxn::commit* and *DbTxn::abort* calls. Note, it is not necessary to transaction protect read-only transactions, unless those transactions require repeatable reads.

The Db access methods will make the appropriate calls into the lock, log and memory pool subsystems in order to guarantee that transaction semantics are applied. When the application is ready to exit, all outstanding transactions should have been committed or aborted. At this point, all open Db files should be closed. Once the Db database files are closed, the Db subsystems should be closed, either explicitly or by destroying the *DbEnv*(3) object.

It is also possible to use the locking, logging and transaction subsystems of Db to provide transaction semantics to objects other than those described by the Db access methods. In these cases, the application will need more explicit customization of the subsystems as well as the development of appropriate data-structure-specific recovery functions.

For example, consider an application that provides transaction semantics to data stored in plain UNIX files accessed using the *read*(2) and *write*(2) system calls. The operations for which transaction protection is desired are bracketed by calls to *DbTxnMgr::begin* and *DbTxn::commit*.

Before data are referenced, the application must make a call to the lock manager, *DbLock*(3), 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. It is up to the application to ensure that appropriate locks are acquired. Before a write is performed, the application should acquire a write lock on the object, by making an appropriate call to the lock manager, *DbLock*(3). Then, the application should make a call to the log manager, *DbLog*, to record enough information to redo the operation in case of failure after commit and to undo the operation in case of abort. As discussed in the *DbLog*(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 identifies the file being modified, what part is redo information, and what part is undo information.

After the log message is written, the application may issue the write system call. After all requests are issued, the application may call *DbTxn::commit*. When *DbTxn::commit* returns, the caller is guaranteed that all necessary log writes have been written to disk.

At any time, the application may call *DbTxn::abort*, which will result in the appropriate calls to the *recover* method to restore the ‘‘database’’ to a consistent pre-transaction state. (The recover method must be able to either re-apply 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 *DbLog* interface to read the log and call the *recover* method to restore the database to a consistent state.

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

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 *DbTxn::prepare* 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), fcntl(2), fflush(3), and strerror(3).

The *DbTxn::commit* method may fail and throw a *DbException*(3)

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

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

[EINVAL]
The transaction was aborted.

The *DbTxn::abort* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DBenv->tx_recover(3), DbLockTab::vec(3), DbLog::get(3), fcntl(2), fflush(3), memset(3), and strerror(3).

[EINVAL]
The transaction was already aborted.

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)