NAME

db – the DB library overview and introduction

DESCRIPTION

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

The DB library does not provide user interfaces, data entry GUI's, SQL support or any of the other standard user-level database interfaces. What it does provide are the programmatic building blocks that allow you to easily embed database-style functionality and support into other objects or interfaces.

ARCHITECTURE

The DB library supports two different models of applications: client-server and embedded.

In the client-server model, a database server is created by writing an application that accepts requests via some form of IPC and issues calls to the DB functions based on those queries. In this model, applications are client programs that attach to the server and issue queries. The client-server model trades performance for protection, as it does not require that the applications share a protection domain with the server, but IPC/RPC is generally slower than a function call. In addition, this model simplifies the creation of network client-server applications.

In the embedded model, an application links the DB library directly into its address space. This provides for faster access to database functionality, but means that the applications sharing log files, lock manager, transaction manager or memory pool manager have the ability to read, write, and corrupt each other's data.

It is the application designer's responsibility to select the appropriate model for their application.

Applications require a single include file, *<db.h>*, which must be installed in an appropriate location on the system.

C++

The C++ classes provide a thin wrapper around the C API, with the major advantages being improved encapsulation and an optional exception mechanism for errors.

The classes and methods are named in a fashion that directly corresponds to structures and functions in the C interface. Likewise, arguments to methods appear in the same order as the C interface, except to remove the explicit ‘‘this’’ pointer. The #defines used for flags are identical between the C and C++ interfaces.

As a rule, each C++ object has exactly one structure from the underlying C API associated with it. The C structure is allocated with each constructor call and deallocated with each destructor call. Thus, the rules the user needs to follow in allocating and deallocating structures are the same between the C and C++ interfaces.

To ensure portability to many platforms, both new and old, we make few assumptions about the C++ compiler and library. For example, we do not expect STL, templates or namespaces to be available. The newest C++ feature used is exceptions, which are used liberally to transmit error information. Even the use of exceptions can be disabled at runtime, by using *DbEnv::set_error_model*() (see *DbEnv*(3)). For a discussion of the exception mechanism, see *DbException*(3).

For the rest of this manual page, C interfaces are listed as the primary reference, and C++ interfaces following parenthetically, e.g., *db_open* (*Db::open*).

JAVA

The Java classes provide a layer around the C API that is almost identical to the C++ layer. The classes and methods are, for the most part identical to the C++ layer. Db constants and #defines are represented as "static final int" values. Errors conditions appear as Java exceptions.

As in C++, each Java object has exactly one structure from the underlying C API associated with it. The Java structure is allocated with each constructor or open call, but is deallocated only when the Java GC does so. Because the timing or ordering of GC is not predictable, the user should take care to do a close() when finished with any object that has such a method.

SUBSYSTEMS

The DB library is made up of five major subsystems, as follows:

Access methods
: The access methods subsystem is made up of general-purpose support for creating and accessing files formatted as B+tree's, hashed files, and fixed and variable length records. These modules are useful in the absence of transactions for processes that need fast, formatted file support. See *db_open*(3) and *db_cursor*(3) (*Db*(3) and *Dbc*(3)) for more information.

Locking
: The locking subsystem is a general-purpose lock manager used by DB. This module is useful in the absence of the rest of the DB package for processes that require a fast, configurable lock manager. See *db_lock*(3) (*DbLockTab*(3) and *DbLock*(3)) for more information.

Logging
: The logging subsystem is the logging support used to support the DB transaction model. It is largely specific to the DB package, and unlikely to be used elsewhere. See *db_log*(3) (*DbLog*(3)) for more information.

Memory Pool
: The memory pool subsystem is the general-purpose shared memory buffer pool used by DB. This module is useful outside of the DB package for processes that require page-oriented, cached, shared file access. See *db_mpool*(3) (*DbMpool*(3) and *DbMpoolFile*(3)) for more information.

Transactions
: The transaction subsystem implements the DB transaction model. It is largely specific to the DB package. See *db_txn*(3) (*DbTxnMgr*(3) and *DbTxn*(3)) for more information.

There are several stand-alone utilities that support the DB environment. They are as follows:

db_archive
: The *db_archive* utility supports database backup, archival and log file administration. See *db_archive*(1) for more information.

db_recover
: The *db_recover* utility runs after an unexpected DB or system failure to restore the database to a consistent state. See *db_recover*(1) for more information.

db_checkpoint
: The *db_checkpoint* utility runs as a daemon process, monitoring the database log and periodically issuing checkpoints. See *db_checkpoint*(1) for more information.

db_deadlock
: The *db_deadlock* utility runs as a daemon process, periodically traversing the database lock structures and aborting transactions when it detects a deadlock. See *db_deadlock*(1) for more information.

db_dump
: The *db_dump* utility writes a copy of the database to a flat-text file in a portable format. See *db_dump*(1) for more information.

db_load
: The *db_load* utility reads the flat-text file produced by *db_dump*, and loads it into a database file. See *db_load*(1) for more information.

db_stat
: The *db_stat* utility displays statistics for databases and database environments. See *db_stat*(1) for more information.

NAMING AND THE DB ENVIRONMENT

The DB application environment is described by the *db_appinit*(3) (*DbEnv*(3)) manual page. The *db_appinit* (*DbEnv::appinit*) function is used to create a consistent naming scheme for all of the subsystems sharing a DB environment. If *db_appinit* (*DbEnv::appinit*) is not called by a DB application, naming is performed as specified by the manual page for the specific subsystem.

DB applications that run with additional privilege should always call the *db_appinit* (*DbEnv::appinit*) function to initialize DB naming for their application. This ensures that the environment variables DB_HOME and TMPDIR will only be used if the application explicitly specifies that they are safe.

ADMINISTERING THE DB ENVIRONMENT

A DB environment consists of a database home directory and all the long-running daemons necessary to ensure continued functioning of DB and its applications. In the presence of transactions, the checkpoint daemon, *db_checkpoint*, must be run as long as there are applications present (see *db_checkpoint*(1) for details). When locking is being used, the deadlock detection daemon, *db_deadlock*, must be run as long as there are applications present (see *db_deadlock*(1) for details). The *db_archive* utility provides information to facilitate log reclamation and creation of database snapshots (see *db_archive*(1) for details). After application or system failure, the *db_recover* utility must be run before any applications are restarted to return the database to a consistent state (see *db_recover*(1) for details).

The simplest way to administer a DB application environment is to create a single ‘‘home’’ directory that houses all the files for the applications that are sharing the DB environment. In this model, the shared memory regions (i.e., the locking, logging, memory pool, and transaction regions) and log files will be stored in the specified directory hierarchy. In addition, all data files specified using relative pathnames will be named relative to this home directory. When recovery needs to be run (e.g., after system or application failure), this directory is specified as the home directory to *db_recover*(1), and the system is restored to a consistent state, ready for the applications to be restarted.

In situations where further customization is desired, such as placing the log files on a separate device, it is recommended that the application installation process create a configuration file named ‘‘DB_CONFIG’’ in the database home directory, specifying the customization. See *db_appinit*(3) (*DbEnv*(3)) for details on this procedure.

The DB architecture does not support placing the shared memory regions on remote filesystems, e.g., the Network File System (NFS) and the Andrew File System (AFS). For this reason, the database home directory must reside on a local filesystem. Databases, log files and temporary files may be placed on remote filesystems, although the application may incur a performance penalty for doing so.

It is important to realize that all applications sharing a single home directory implicitly trust each other. They have access to each other’s data as it resides in the shared memory buffer pool and will share resources such as buffer space and locks. At the same time, any applications that access the same files **must** share an environment if consistency is to be maintained across the different applications.

ERROR RETURNS

Except for the historic *dbm* and *hsearch* interfaces (see *db_dbm*(3) and *db_hsearch*(3)), DB does not use the global variable *errno* to return error values. The return values for all DB functions can be grouped into three categories:

0 A return value of 0 indicates that the operation was successful.

>0 A return value that is greater than 0 indicates that there was a system error. The *errno* value returned by the system is returned by the function, e.g., when a DB function is unable to allocate memory, the return value from the function will be ENOMEM.

<0 A return value that is less than 0 indicates a condition that was not a system failure, but was not an unqualified success, either. For example, a routine to retrieve a key/data pair from the database may return DB_NOTFOUND when the key/data pair does not appear in the database, as opposed to the value of 0, which would be returned if the key/data pair were found in the database. All such special values returned by DB functions are less than 0 in order to avoid conflict with possible values of *errno*.

There are two special return values that are somewhat similar in meaning, are returned in similar situations, and therefore might be confused: DB_NOTFOUND and DB_KEYEMPTY. The DB_NOTFOUND error return indicates that the requested key/data pair did not exist in the database or that start- or end-of-file has been reached. The DB_KEYEMPTY error return indicates that the requested key/data pair logically exists but was never explicitly created by the application (the recno access method will automatically create key/data pairs under some circumstances, see *db_open*(3) (*Db*(3)) for more information), or that the requested key/data pair was deleted and is currently in a deleted state.

SIGNALS

When applications using DB receive signals, it is important that they exit gracefully, discarding any DB locks that they may hold. This is normally done by setting a flag when a signal arrives, and then checking for that flag periodically within the application. Specifically, the signal handler should not attempt to release locks and/or close the database handles itself. This is not guaranteed to work correctly and the results are undefined.

If an application exits holding a lock, the situation is no different than if the application crashed, and all applications participating in the database environment must be shutdown, and then recovery must be performed. If this is not done, the locks that the application held can cause unresolvable deadlocks inside the database, and applications may then hang.

MULTI-THREADING

See *db_thread*(3) for information on using DB in threaded applications.

DATABASE AND PAGE SIZES

DB stores database file page numbers as unsigned 32-bit numbers and database file page sizes as unsigned 16-bit numbers. This results in a maximum database size of 2^48. The minimum database page size is 512 bytes, resulting in a minimum maximum database size of 2^41.

DB is potentially further limited if the host system does not have filesystem support for files larger than 2^32, including seeking to absolute offsets within such files.

The maximum btree depth is 255.

BYTE ORDERING

The database files created by DB can be created in either little or big-endian formats. By default, the native format of the machine on which the database is created will be used. Any format database can be used on a machine with a different native format, although it is possible that the application will incur a performance penalty for the run-time conversion.

EXTENDING DB

DB includes tools to simplify the development of application-specific logging and recovery. Specifically, given a description of the information to be logged, these tools will automatically create logging functions (functions that take the values as parameters and construct a single record that is written to the log), read functions (functions that read a log record and unmarshall the values into a structure that maps onto the values you chose to log), a print function (for debugging), templates for the recovery functions, and automatic dispatching to your recovery functions.

EXAMPLES

There are a number of examples included with the DB library distribution, intended to demonstrate various ways of using the DB library.

Some applications require the use of formatted files to store data, but do not require concurrent access and can cope with the loss of data due to catastrophic failure. Generally, these applications create short-lived databases that are discarded or recreated when the system fails. Such applications need only use the DB access methods. The DB access methods will use the memory pool subsystem, but the application is unlikely to do so explicitly. See the files *examples/ex_access.c*, *examples/ex_btrec.c*, *examples_cxx/AccessExample.cpp* and *java/src/com/sleepycat/examples/AccessExample.java* in the DB source distribution for C, C++, and Java language code examples of how such applications might use the DB library.

Some applications require the use formatted files to store data, but also need to use *db_appinit*(3) (*DbEnv::appinit*(3)) for environment initialization. See the files *examples/ex_appinit.c*, *examples_cxx/AppinitExample.cpp* or *java/src/com/sleepycat/examples/AppinitExample.java* in the DB source distribution for C, C++ and Java language code examples of how such an application might use the DB library.

Some applications use the DB access methods, but are also concerned about catastrophic failure, and therefore need to transaction protect the underlying DB files. See the files *examples/ex_tpcb.c*, *examples_cxx/TpcbExample.cpp* or *java/src/com/sleepycat/examples/TpcbExample.java* in the DB source distribution for C, C++ and Java language code examples of how such an application might use the DB library.

Some applications will benefit from the ability to buffer input files other than the underlying DB access method files. See the files *examples/ex_mpool.c* or *examples_cxx/MpoolExample.cpp* in the DB source distribution for C and C++ language code examples of how such an application might use the DB library.

Some applications need a general-purpose lock manager separate from locking support for the DB access methods. See the files *examples/ex_lock.c*, *examples_cxx/LockExample.cpp* or *java/src/com/sleepycat/examples/LockExample.java* in the DB source distribution for C, C++ and Java language code examples of how such an application might use the DB library.

Some applications will use the DB access methods in a threaded fashion, including trickle flushing of the underlying buffer pool and deadlock detection. See the file *examples/ex_thread.c* in the DB source distribution for a C language code example of how such an application might use the DB library. Note that the Java API assumes a threaded environment and performs all thread-specific initialization automatically.

COMPATIBILITY

The DB 2.0 library provides backward compatible interfaces for the historic UNIX *dbm*(3), *ndbm*(3) and *hsearch*(3) interfaces. See *db_dbm*(3) and *db_hsearch*(3) for further information on these interfaces. It also provides a backward compatible interface for the historic DB 1.85 release. **DB 2.0 does not provide database compatibility for any of the above interfaces, and existing databases must be converted manually.** To convert existing databases from the DB 1.85 format to the DB 2.0 format, review the *db_dump185*(1) and *db_load*(1) manual pages.

The name space in DB 2.0 has been changed from that of previous DB versions, notably version 1.85, for portability and consistency reasons. The only name collisions in the two libraries are the names used by the *dbm*(3), *ndbm*(3), *hsearch*(3) and the DB 1.85 compatibility interfaces. To include both DB 1.85 and DB 2.0 in a single library, remove the *dbm*(3), *ndbm*(3) and *hsearch*(3) interfaces from either of the two libraries, and the DB 1.85 compatibility interface from the DB 2.0 library. This can be done by editing the library Makefiles and reconfiguring and rebuilding the DB 2.0 library. Obviously, if you use the historic interfaces, you will get the version in the library from which you did not remove it. Similarly, you will not be able to access DB 2.0 files using the DB 1.85 compatibility interface, since you have removed that from the library as well.

It is possible to simply relink applications written to the DB 1.85 interface against the DB 2.0 library. Recompilation of such applications is slightly more complex. When the DB 2.0 library is installed, it installs two include files, *db.h* and *db_185.h*. The former file is likely to replace the DB 1.85 version's include file which had the same name. If this did not happen, recompiling DB 1.85 applications to use the DB 2.0 library is simple: recompile as done historically, and load against the DB 2.0 library instead of the DB 1.85 library. If, however, the DB 2.0 installation process has replaced the system's *db.h* include file, replace the application's include of *db.h* with inclusion of *db_185.h*, recompile as done historically, and then load against the DB 2.0 library.

Applications written using the historic interfaces of the DB library should not require significant effort to port to the DB 2.0 interfaces. While the functionality has been greatly enhanced in DB 2.0, the historic interface and functionality and is largely unchanged. Reviewing the application's calls into the DB library and updating those calls to the new names, flags and return values should be sufficient.

While loading applications that use the DB 1.85 interfaces against the DB 2.0 library, or converting DB 1.85 function calls to DB 2.0 function calls will work, reconsidering your application's interface to the DB database library in light of the additional functionality in DB 2.0 is recommended, as it is likely to result in enhanced application performance.

SEE ALSO: ADMINISTRATIVE AND OTHER UTILITIES

db_archive(1), *db_checkpoint*(1), *db_deadlock*(1), *db_dump*(1), *db_load*(1), *db_recover*(1), *db_stat*(1)

SEE ALSO: C API

db_appinit(3), *db_cursor*(3), *db_dbm*(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_txn*(3)

SEE ALSO: C++ and Java API

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)

SEE ALSO: ADDITIONAL REFERENCES

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