NAME

tdbm – dbm database functions with nested atomic transactions

SYNOPSIS

```
#include <tdbm.h>

DbmRc DbmAbort(tid)
Tid *tid;

DbmRc DbmBegin(parent, child)
Tid *parent;
Tid **child;

DbmRc DbmClose(dbm)
Dbm *dbm;

DbmRc DbmCommit(tid)
Tid *tid;

DbmRc DbmDelete(dbm, tid, key)
Dbm *dbm;
Tid *tid;
Datum key;

char *DbmErrorString(rc)
DbmRc rc;

DbmRc DbmFetch(dbm, tid, key, value)
Dbm *dbm;
Tid *tid;
Datum key;
Datum *value;

DbmRc DbmFind(dbm, tid, key, value)
Dbm *dbm;
Tid *tid;
Datum key;
Datum *value;

DbmRc DbmFirstkey(dbm, tid, key)
Dbm *dbm;
Tid *tid;
Datum *key;

DbmRc DbmNextkey(dbm, tid, key)
Dbm *dbm;
Tid *tid;
Datum *key;

DbmRc DbmOpen(pathname, type, config, dbm, recovery)
char *pathname;
DbmFileType type;
DbmConfig *config;
Dbm **dbm;
DbmRecovery **recovery;
```

```
DbmRc DbmPrecommit(tid, tidname)
Tid *tid;
DbmTidName *tidname;

DbmErrorClass DbmRcClass(rc)
DbmRc rc;

DbmRc DbmRecover(dbm, recovery, tid)
Dbm *dbm;
DbmRecovery **recovery;
Tid **tid;

DbmRc DbmStore(dbm, tid, key, value, mode)
Dbm *dbm;
Tid *tid;
Datum key;
Datum value;
DbmStoreMode mode;
```

DESCRIPTION

Tdbm is a collection of functions that implement a simple database made up of key/content pairs. While similar to the UNIX **dbm**(3) and **ndbm**(3) packages, there are a number of significant differences:

- Nested atomic transactions are supported across a single database.
- Volatile (temporary and memory resident) databases can be used.
- A database is implemented as a single file.
- Very large objects can be stored.
- In a multi-threaded environment, concurrent transactions are possible.
- In many cases, performance should be improved.

The usual **Datum** data structure has an additional component, a datum descriptor:

```
typedef u_char EntryDesc;

typedef struct {
  char        *dptr;
  int         dsize;
  EntryDesc   desc;
} Datum;
```

The descriptor can be used to specify the alignment requirements of a key or value. For the value, the system preserves the requested alignment when it is read so that it can be accessed directly from the system buffer, obviating the need to copy the value into properly aligned memory. For the key, the alignment is not preserved but the specified alignment can be determined later.

The low-order two bits of the descriptor specify alignment. The following constants are defined:

ALIGN0 - No alignment required

ALIGN2 - Align on any even address

ALIGN4 - Align on any address divisible by 4

ALIGN8 - Align on any address divisible by 8

Before a database can be used, it must be opened by calling **DbmOpen( ).** The **pathname** argument identifies the database to be used; it is created if necessary. If **type** is **DBM_PERSISTENT**, then the database will be a normal Unix file. The file will be exclusively locked using **flock**(2). If it is **DBM_VOLATILE**, then the database is temporary and will disappear when it is no longer referenced (or when the program terminates). A database can be opened multiple times except for one special case. For volatile databases, if **pathname** is NULL or the null string, then a unique internal name is effectively assigned to the database.

If not NULL, **config** allows the user to override system defaults for database parameters: If **recovery** is not NULL, a list of precommitted transactions is returned (see **DbmPrecommit( )** and **DbmRecover( )**).

```
typedef struct DbmConfig {
  int        mode;        /* Mode when creating new dbm */
  int        pagesize;    /* Size of each page (bucket) in the dbm */
  int        allocunits;  /* Allocation units, w.r.t. pagesize */
} DbmConfig;

typedef struct DbmTidName {
  int        hostid;
  int        start_time;
  int        count;
} DbmTidName;

typedef struct DbmRecovery {
  char        *pathname;
  DbmTidName  tidname;
  struct DbmRecovery *next;
} DbmRecovery;
```

A default is overridden only if a configuration option is non-zero. An identifier associated with the opened database is returned.

Most operations on a database occur within the context of a transaction. A transaction is initiated by calling **DbmBegin( ).** If the **parent** argument is NULL, then this is a top-level transaction, otherwise the new transaction is a subtransaction of **parent**. A new transaction identifier **tid** is returned. In the current implementation, all operations within a top-level transaction must be associated with a single database; a transaction is implicitly associated with a database based on the first I/O operation it performs.

N.B. The value (or key) returned by a function must be copied if it is going to be modified. Also, a datum's **dptr** may no longer be valid if the transaction with which it is associated aborts.

New entries are stored in the database using **DbmStore( ).** The given **value** is put into the database using **key**. The **mode** argument is either **DBM_REPLACE** or **DBM_INSERT**. The former deletes an existing instance of an entry with the same key before storing the new entry while the latter insists that there be no existing entry. The **key** and **value** are copied, so they may be freed after this call.

DbmFetch( ) is used to retrieve an entry. **DbmFind( )** locates an entry without retrieving the value. Both functions set the descriptor for the key and the value.

An entry is deleted from the database by calling **DbmDelete( )**.

A database can be traversed in (apparently) random order. **DbmFirstkey( )** retrieves the key of the ‘‘first’’ entry in the database. **DbmNextkey( )** may be called to retrieve the keys of successive entries. Both functions set the descriptor for the key. Note that these functions should be used carefully since they could end up locking the entire database while their transaction exists.

A transaction is committed by calling **DbmCommit( )** or aborted by calling **DbmAbort( ).** Aborting a transaction rolls back all modifications made to the database by the transaction. In either case, the transaction must not have any subtransactions.

A top-level transaction may be prepared for commitment, but not actually committed, by calling **DbmPrecommit( )**. Upon successful completion, the system guarantees that a subsequent commit will succeed (modulo media failures). After this call, the only operations allowed on the transaction are **DbmCommit( )** and **DbmAbort( )**. If **tidname** is not NULL, then it is initialized to the (globally unique) name of the transaction. If **DbmOpen( )** returns recovery information, then one or more precommitted transactions exist. **DbmRecover( )** should be called (until ***recovery** is NULL) to return a transaction identifier and the transaction name for each precommitted transaction. These functions are expected to be used by a two phase commit protocol.

A database is closed using **DbmClose( ).** Closing a database aborts any active transactions using it. The given **dbm** identifier is invalidated, yet if the database was opened multiple times, operations on the database may continue using the other identifiers.

DbmErrorString( ) returns an error message corresponding to the given result code. **DbmRcClass( )** returns an indication of the type of error that has occurred (e.g., **DBM_FATAL**).

WARNING

It is unwise for a program to update a database that it is accessing via NFS; databases should always be accessed directly on the file’s server. NFS file locking is not performed. Also, byte ordering is not addressed by the current implementation.

FILES

As part of the atomic commit operation, a transaction file is created. Its name is that of the database with a ‘‘**.trans**’’ appended. The mode of this file has special meaning to *tdbm* and therefore should not be changed by the user.

SEE ALSO

dbm(3), **ndbm**(3).

The GNU gdbm library, Philip A. Nelson <phil@cs.wwu.edu>, Computer Science Department, Western Washington University.

The Berkeley Hash package, Margo Seltzer <margo@postgres.berkeley.edu>. See: ‘‘A New Hash Package for UNIX’’ by Margo Seltzer and Ozan Yigit in Winter 1991 Usenix Proceedings.

An interactive program, **tdbm**(1), is available to manipulate and inspect **tdbm** files.

DIAGNOSTICS

When an unusual error occurs, a message is likely to be printed on **stderr**.

BUGS

Because creation of a new database occurs outside of the transaction mechanism, rollback is not possible.

Multithreaded operation is currently supported only for the U.B.C. Threads kernel.

There are currently configuration-dependent limits on the length of a **key** (it must fit into a database ‘‘page’’) and the length of a **value** (depends on the size of the page allocation bit map and other factors).

Although very large, there are limits on the maximum size of a database. A database cannot span filesystems (unless the O/S provides it transparently).

A mechanism to support efficient tree locking should be added.

The space used by deleted entries is not reclaimed by the operating system but may be reused by the database. The database file may contain holes, making it appear much bigger that it really is. There is no automatic ‘‘shrinking’’ of the database. If desired, this must be done by traversing the old database, copying each entry to a new database.

The package cannot recover from media failures.

The location of the transaction file is not user configurable.

The obvious method of deleting (or adding) items while iterating through the database using **DbmFirstKey( ), DbmNextKey( ),** is buggy. No updates should be performed until you’re done iterating.

The wish list of additional functionality is too long (but suggestions and bug reports are welcome). The following user-visible improvements currently top the list:

- Deadlock detection (eg., via timestamps).
- Performance instrumentation ‘‘**DbmStats( )**’’.
- Make **DbmRc** a structure that includes **errno**.
- Arbitrary size keys.
- Multi-volume dbms (e.g., make it possible to allocate pages on a different filesystem once a limit is reached (soft or hard)).
- Multi-dbm transactions.

AUTHOR

Barry Brachman <brachman@cs.ubc.ca>
Dept. of Computer Science
University of British Columbia