NAME

Dbt − Db key/data class

SYNOPSIS

void *Dbt::get_data() const;
void Dbt::set_data(void *);

u_int32_t Dbt::get_size() const;
void Dbt::set_size(u_int32_t);

u_int32_t Dbt::get_ulen() const;
void Dbt::set_ulen(u_int32_t);

u_int32_t Dbt::get_dlen() const;
void Dbt::set_dlen(u_int32_t);

u_int32_t Dbt::get_doff() const;
void Dbt::set_doff(u_int32_t);

u_int32_t Dbt::get_flags() const;
void Dbt::set_flags(u_int32_t);

Dbt::Dbt(void *data, size_t size);
Dbt::Dbt();
Dbt::~Dbt();
Dbt::Dbt(const Dbt &);
Dbt::Dbt &operator = (const Dbt &);

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 *Dbt* class, used to encode keys and data items in a database.

KEY/DATA PAIRS

Storage and retrieval for the Db access methods are based on key/data pairs. Both key and data items are represented by Dbt objects.

Key and data byte strings may reference strings of essentially unlimited length, although any two keys must fit into available memory at the same time so that they may be compared, and any one data item must fit into available memory so that it may be returned.

The Dbt class provides simple access to an underlying data structure, whose elements can be examined or changed using the *set_* or *get_* methods. The remainder of the manual page sometimes refers to these accesses using the underlying name, e.g., simply *ulen* instead of *get_ulen* and *set_ulen*. The constructors set all elements of the underlying structure to zero. The constructor with two arguments has the effect of setting all elements to zero except for the specified *data* and *size* elements. In the case where the *flags* structure element is 0, when being provided a key or data item by the application, the DB package expects the *data* object to point to a byte string of *size* bytes. When returning a key/data item to the application, the DB package will store into the *data* object a pointer to a byte string of *size* bytes. **By default, the memory referenced by this stored pointer is only valid until the next call to the DB package using the Db handle returned by** *Db::open*.

The access methods provide no guarantees about byte string alignment, and applications are responsible for maintaining any necessary alignment. Use the DB_DBT_USERMEM flag to cause returned items to be placed in memory of arbitrary alignment. Although Java normally maintains proper alignment of byte arrays, the set_offset method can be used to specify unaligned addresses. Unaligned address accesses that are not supported by the underlying hardware may be reported as an exception, or may stop the running Java program.

The elements of the structure underlying the Dbt class are defined as follows:

void *data;

A pointer to a byte string. This element is accessed using *get_data* and *set_data*, and may be initialized using one of the constructors.

int offset;

The number of bytes offset into the *data* array to determine the portion of the array actually used. This element is accessed using *get_offset* and *set_offset*.

u_int32_t size;

The length of *data*, in bytes. This element is accessed using *get_size* and *set_size*, and may be initialized using the constructor with two arguments.

u_int32_t ulen;

The size of the user's buffer (referenced by *data*), in bytes. This location is not written by the Db functions. See the DB_DBT_USERMEM flag for more information. This element is accessed using *get_ulen* and *set_ulen*.

u_int32_t dlen;

The length of the partial record being read or written by the application, in bytes. See the DB_DBT_PARTIAL flag for more information. This element is accessed using *get_dlen* and *set_dlen*.

u_int32_t doff;

The offset of the partial record being read or written by the application, in bytes. See the DB_DBT_PARTIAL flag for more information. This element is accessed using *get_doff* and *set_doff*.

u_int32_t flags;

This element is accessed using *get_flags* and *set_flags*. The flags value is specified by **or**'ing together one or more of the following values:

DB_DBT_MALLOC

Ignored except when retrieving information from a database, e.g., a *Db::get* or *Dbc::get* call. This flag causes Db to allocate memory for the returned key or data item (using *malloc*(3), or the user-specified malloc method) and return a pointer to it in the *data* field of the key or data Dbt object. The allocated memory becomes the responsibility of the calling application. It is an error to specify both DB_DBT_MALLOC and DB_DBT_USERMEM.

DB_DBT_USERMEM

Ignored except when retrieving information from a database, e.g., a *Db::get* or *Dbc::get* call. The *data* field of the key or data object must reference memory that is at least *ulen* bytes in length. If the length of the requested item is less than or equal to that number of bytes, the item is copied into the memory referenced by the *data* field. Otherwise, an error is returned, the *size* field is set to the length needed for the requested item, and the *errno* variable is set to ENOMEM. It is an error to specify both DB_DBT_MALLOC and DB_DBT_USERMEM.

DB_DBT_PARTIAL

Ignored except when specified for a data parameter, where this flag causes the partial retrieval or storage of an item. If the calling application is doing a get, the *dlen* bytes starting *doff* bytes from the beginning of the retrieved data record are returned as if they comprised the entire record. If any or all of the specified bytes do not exist in the record, the get is successful and the existing bytes or 0 bytes are returned.

For example, if the data portion of a retrieved record was 100 bytes, and a partial retrieval was done using a Dbt having a *dlen* field of 20 and a *doff* field of 85, the get call would succeed, the *data* field would reference the last 15 bytes of the record, and the *size* field would be set to 15.

If the calling application is doing a put, the *dlen* bytes starting *doff* bytes from the beginning of the specified key's data record are replaced by the data specified by the *data* and *size* objects. If *dlen* is smaller than *size*, the record will grow, and if *dlen* is larger than *size*, the record will shrink. If the specified bytes do not exist, the record will be extended using nul bytes as necessary, and the put call will succeed.

It is an error to attempt a partial put using the *Db::put* method in a database that supports duplicate records. Partial puts in databases supporting duplicate records must be done using a *Db::cursor* method. It is an error to attempt a partial put with differing *dlen* and *size* values in a recno database with fixed-length records.

For example, if the data portion of a retrieved record was 100 bytes, and a partial put was done using a Dbt having a *dlen* field of 20, a *doff* field of 85, and a *size* field of 30, the resulting record would be 115 bytes in length, where the last 30 bytes would be those specified by the put call.

The default algorithm of associating returned key or data items with the Db handle returned by *Db::open*(3) will obviously not work when Db handles are being used concurrently by multiple threads within a process, i.e, when DB_THREAD was specified to *Db::open*(3). **When multiple threads are using the returned Db handle concurrently, either the DB_DBT_MALLOC or DB_DBT_USERMEM flags must be specified for any Dbt used for key or data retrieval.**

LOGICAL RECORD NUMBERS

In all cases for the recno access method, and when calling the *Db::get* and *Dbc::get* functions with the DB_GET_RECNO flag specified, the *data*

field of the key must be a pointer to a memory location of type *db_recno_t*, as typedef'd in the <db_cxx.h> include file. This type is a 32-bit unsigned type, (which limits the number of logical records in a recno database, and the maximum logical record which may be directly retrieved from a btree database, to 4,294,967,296). The *size* field of the key should be the size of that type, e.g.,

in the C programming language, ‘‘sizeof(db_recno_t)’’.

Logical record numbers are 1-based, not 0-based, i.e., the first record in the database is record number 1.

BUGS

The Db access methods provide no guarantees about byte string alignment, and applications are responsible for maintaining any necessary alignment.

The name Dbt is a mnemonic for ‘‘data base thang’’, and was used because noone could think of a reasonable name that wasn't already in use somewhere else.

SEE ALSO

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)