NAME

DbInfo – informational parameters for database open

SYNOPSIS

```
#include <db_cxx.h>

DbInfo::DbInfo();
DbInfo::~DbInfo();
DbInfo::DbInfo(const DbInfo &);
DbInfo::DbInfo &operator = (const DbInfo &);

int DbInfo::get_lorder() const;
void DbInfo::set_lorder(int);

size_t DbInfo::get_cachesize() const;
void DbInfo::set_cachesize(size_t);

size_t DbInfo::get_pagesize() const;
void DbInfo::set_pagesize(size_t);

typedef void *(*db_malloc_fcn)(size_t);
DbInfo::db_malloc_fcn DbInfo::get_malloc() const;
void DbInfo::set_malloc(db_malloc_fcn);

int DbInfo::get_bt_maxkey() const;
void DbInfo::set_bt_maxkey(int);

int DbInfo::get_bt_minkey() const;
void DbInfo::set_bt_minkey(int);

typedef int (*bt_compare_fcn)(const DBT *, const DBT *);
bt_compare_fcn DbInfo::get_bt_compare() const;
void DbInfo::set_bt_compare(bt_compare_fcn);

typedef size_t (*bt_prefix_fcn)(const DBT *, const DBT *);
bt_prefix_fcn DbInfo::get_bt_prefix() const;
void DbInfo::set_bt_prefix(bt_prefix_fcn);

unsigned int DbInfo::get_h_ffactor() const;
void DbInfo::set_h_ffactor(unsigned int);

unsigned int DbInfo::get_h_nelem() const;
void DbInfo::set_h_nelem(unsigned int);

typedef u_int32_t (*h_hash_fcn)(const void *, u_int32_t);
h_hash_fcn DbInfo::get_h_hash() const;
void DbInfo::set_h_hash(h_hash_fcn);

int DbInfo::get_re_pad() const;
void DbInfo::set_re_pad(int);

int DbInfo::get_re_delim() const;
void DbInfo::set_re_delim(int);

u_int32_t DbInfo::get_re_len() const;
void DbInfo::set_re_len(u_int32_t);

char DbInfo::*get_re_source() const;
void DbInfo::set_re_source(char *);
```

u_int32_t DbInfo::get_flags() const;
void DbInfo::set_flags(u_int32_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 DbInfo class. A DbInfo object is used in conjunction with the *Db::open* method (see *Db*(3)) to specify particular configuration options for the open. The DbInfo 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 refers to these accesses using the underlying name, e.g., *cachesize* instead of *get_cachesize* and *set_cachesize*. The default constructor sets all elements of the underlying structure to zero. Some of the fields are specific to a type of file format (one of btree, hashed and recno) and are thus named with an underscore separated string, ''bt'', ''h'' and ''re'', respectively. For example, the method *set_bt_maxkey* sets the underlying *bt_maxkey* field, and this field is only used when opening a btree file.

The fields that are common to all access methods are listed here; those specific to an individual access method are described below. No reference to the DbInfo object is maintained by Db, so it is possible to discard it as soon as the *Db::open* call returns.

If possible, defaults appropriate for the system are used for the DbInfo fields if *dbinfo* is NULL or any fields of the DbInfo object are not explicitly set. The following DbInfo fields may be initialized before calling *Db::open*:

size_t cachesize;

A suggested maximum size of the memory pool cache, in bytes. If *cachesize* is not explicitly set, an appropriate default is used. If the *mp_info* field is also specified, this field is ignored.

Note, the minimum number of pages in the cache should be no less than 10, and the access methods will fail if an insufficiently large cache is specified. In addition, for applications that exhibit strong locality in their data access patterns, increasing the size of the cache can significantly improve application performance.

int lorder;

The byte order for integers in the stored database metadata. The number should represent the order as an integer, for example, big endian order is the number 4,321, and little endian order is the number 1,234. If *lorder* is not explicitly set, the host order of the machine where the Db library was compiled is used.

The value of *lorder* is ignored except when databases are being created. If a database already exists, the byte order it uses is determined when the file is read.

The access methods provide no guarantees about the byte ordering of the application data stored in the database, and applications are responsible for maintaining any necessary ordering.

size_t pagesize;

The size of the pages used to hold items in the database, in bytes. The minimum page size is 512 bytes and the maximum page size is 64K bytes. If *pagesize* is not explicitly set, a page size is selected based on the underlying filesystem I/O block size. The selected size has a lower limit of 512 bytes and an upper limit of 16K bytes.

void *(*malloc)(size_t);

The flag DB_DBT_MALLOC, when specified in the Dbt object, will cause the Db library to allocate memory which then becomes the responsibility of the calling application. See *Dbt*(3) for more

information.

On systems where separate heaps are maintained for applications and libraries (notably Windows NT), specifying the DB_DBT_MALLOC flag will fail because the Db library will allocate memory from a different heap than the application will use to free it. To avoid this problem, the *malloc* function should be set to point to the application's allocation routine. If *malloc* is not explicitly set, it will be used to allocate the memory returned when the DB_DBT_MALLOC flag is set. The *malloc* method must match the calling conventions of the *malloc*(3) library routine.

BTREE

The btree data structure is a sorted, balanced tree structure storing associated key/data pairs. Searches, insertions, and deletions in the btree will all complete in O (lg base N) where base is the average number of keys per page. Often, inserting ordered data into btrees results in pages that are half-full. This implementation has been modified to make ordered (or inverse ordered) insertion the best case, resulting in nearly perfect page space utilization.

Space freed by deleting key/data pairs from the database is never reclaimed from the filesystem, although it is reused where possible. This means that the btree storage structure is grow-only. If sufficiently many keys are deleted from a tree that shrinking the underlying database file is desirable, this can be accomplished by creating a new tree from a scan of the existing one.

The following additional fields and flags may be initialized in the DbInfo object before calling *Db::open*, when using the btree access method:

int (*bt_compare)(const Dbt *, const Dbt *);

The *bt_compare* function is the key comparison method. It must return an integer less than, equal to, or greater than zero if the first key argument is considered to be respectively less than, equal to, or greater than the second key argument. The same comparison method must be used on a given tree every time it is opened. If *bt_compare* is NULL, the keys are compared lexically, with shorter keys collating before longer keys.

int bt_minkey;

The minimum number of keys that will be stored on any single page. This value is used to determine which keys will be stored on overflow pages, i.e. if a key or data item is larger than the pagesize divided by the *bt_minkey* value, it will be stored on overflow pages instead of in the page itself. The *bt_minkey* value specified must be at least 2; if *bt_minkey* is not explicitly set, a value of 2 is used.

size_t (*bt_prefix)(const Dbt *, const Dbt *);

The *bt_prefix* function is the prefix comparison method. If specified, this method must return the number of bytes of the second key argument that are necessary to determine that it is greater than the first key argument. If the keys are equal, the key length should be returned.

This is used to compress the keys stored on the btree internal pages. The usefulness of this is data dependent, but in some data sets can produce significantly reduced tree sizes and search times. If *bt_prefix* is not explicitly set, and no comparison method is specified, a default lexical comparison method is used. If *bt_prefix* is NULL and a comparison method is specified, no prefix comparison is done.

unsigned long flags;

The following additional flags may be specified:

DB_DUP

Permit duplicate keys in the tree, i.e. insertion when the key of the key/data pair being inserted already exists in the tree will be successful. The ordering of duplicates in the tree is determined by the order of insertion, unless the ordering is otherwise specified by use of a cursor (see *Dbc*(3) for more information.) It is an error to specify both DB_DUP and DB_RECNUM.

DB_RECNUM

Support retrieval from btrees using record numbers. For more information, see the

DB_GET_RECNO flag to the *db->get* method (below), and the cursor *Dbc::get* method (in *Dbc*(3)).

Logical record numbers in btrees are mutable in the face of record insertion or deletion. See the DB_RENUMBER flag in the RECNO section below for further discussion.

Maintaining record counts within a btree introduces a serious point of contention, namely the page locations where the record counts are stored. In addition, the entire tree must be locked during both insertions and deletions, effectively single-threading the tree for those operations. Specifying DB_RECNUM can result in serious performance degradation for some applications and data sets.

It is an error to specify both DB_DUP and DB_RECNUM.

HASH

The hash data structure is an extensible, dynamic hashing scheme. Backward compatible interfaces to the functions described in *dbm*(3), *ndbm*(3) and *hsearch*(3) are provided, however these interfaces are not compatible with previous file formats.

The following additional fields and flags may be initialized in the DbInfo object before calling *Db::open*, when using the hash access method:

unsigned int h_ffactor;

The desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. The default value is 0, indicating that the fill factor will be selected dynamically as pages are filled.

u_int32_t (*h_hash)(const void *, u_int32_t);

The *h_hash* field is a user defined hash method; if *h_hash* is NULL, a default hash method is used. Since no hash method performs equally well on all possible data, the user may find that the built-in hash method performs poorly with a particular data set. User specified hash functions must take a pointer to a byte string and a length as arguments and return a u_int32_t value.

If a hash method is specified, *hash_open* will attempt to determine if the hash method specified is the same as the one with which the database was created, and will fail if it detects that it is not.

unsigned int h_nelem;

An estimate of the final size of the hash table. If not set or set too low, hash tables will expand gracefully as keys are entered, although a slight performance degradation may be noticed. The default value is 1.

unsigned long flags;

The following additional flags may be specified by **or**'ing together one or more of the following values:

DB_DUP

Permit duplicate keys in the tree, i.e. insertion when the key of the key/data pair being inserted already exists in the tree will be successful. The ordering of duplicates in the tree is determined by the order of insertion, unless the ordering is otherwise specified by use of a cursor (see *Dbc*(3) for more information.)

RECNO

The recno access method provides support for fixed and variable length records, optionally backed by a flat text (byte stream) file. Both fixed and variable length records are accessed by their logical record number.

It is valid to create a record whose record number is more than one greater than the last record currently in the database. For example, the creation of record number 8, when records 6 and 7 do not yet exist, is not an error. However, any attempt to retrieve such records (e.g., records 6 and 7) will return DB_KEYEMPTY.

Deleting a record will not, by default, renumber records following the deleted record (see DB_RENUMBER below for more information). Any attempt to retrieve deleted records will return DB_KEYEMPTY.

The following additional fields and flags may be initialized in the DbInfo object before calling *Db::open*, when using the recno access method:

int re_delim;
: For variable length records, if the *re_source* file is specified and the DB_DELIMITER flag is set, the delimiting byte used to mark the end of a record in the source file. If the *re_source* file is specified and the DB_DELIMITER flag is not set, <newline> characters (i.e. "\n", 0x0a) are interpreted as end-of-record markers.

u_int32_t re_len;
: The length of a fixed-length record.

int re_pad;
: For fixed length records, if the DB_PAD flag is set, the pad character for short records. If the DB_PAD flag is not explicitly set, <space> characters (i.e., 0x20) are used for padding.

char *re_source;
: The purpose of the *re_source* field is to provide fast access and modification to databases that are normally stored as flat text files.

  If the *re_source* field is explicitly set, it specifies an underlying flat text database file that is read to initialize a transient record number index. In the case of variable length records, the records are separated by the byte value *re_delim*. For example, standard UNIX byte stream files can be interpreted as a sequence of variable length records separated by <newline> characters.

  In addition, when cached data would normally be written back to the underlying database file (e.g., the *close* or *sync* functions are called), the in-memory copy of the database will be written back to the *re_source* file.

  By default, the backing source file is read lazily, i.e., records are not read from the file until they are requested by the application. **If multiple processes (not threads) are accessing a recno database concurrently and either inserting or deleting records, the backing source file must be read in its entirety before more than a single process accesses the database, and only that process should specify the backing source file as part of the** *Db::open* call. See the DB_SNAPSHOT flag below for more information.

  Reading and writing the backing source file specified by re_source cannot be transactionally protected because it involves filesystem operations that are not part of the Db transaction methodology. For this reason, if a temporary database is used to hold the records, i.e., a NULL was specified as the *file* argument to *Db::open*, it is possible to lose the contents of the *re_source* file, e.g., if the system crashes at the right instant. If a file is used to hold the database, i.e., a file name was specified as the *file* argument to *Db::open*, normal database recovery on that file can be used to prevent information loss, although it is still possible that the contents of *re_source* will be lost if the system crashes.

  The *re_source* file must already exist (but may be zero-length) when *Db::open* is called.

  For all of the above reasons, the *re_source* field is generally used to specify databases that are read-only for Db applications, and that are either generated on the fly by software tools, or modified using a different mechanism, e.g., a text editor.

unsigned long flags;
: The following additional flags may be specified by **or**'ing together one or more of the following values:

  DB_DELIMITER
  : The *re_delim* field is set.

DB_FIXEDLEN
The records are fixed-length, not byte delimited. The *re_len* value specifies the length of the record, and the *re_pad* value is used as the pad character.

Any records added to the database that are less than *re_len* bytes long are automatically padded. Any attempt to insert records into the database that are greater than *re_len* bytes long will cause the call to fail immediately and return an error.

DB_PAD
The *re_pad* field is set.

DB_RENUMBER
Specifying the DB_RENUMBER flag causes the logical record numbers to be mutable, and change as records are added to and deleted from the database. For example, the deletion of record number 4 causes records numbered 5 and greater to be renumbered downward by 1. If a cursor was positioned to record number 4 before the deletion, it will reference the new record number 4, if any such record exists, after the deletion. If a cursor was positioned after record number 4 before the deletion, it will be shifted downward 1 logical record, continuing to reference the same record as it did before.

Using the *Dbc::put* or *put* interfaces to create new records will cause the creation of multiple records if the record number is more than one greater than the largest record currently in the database. For example, creating record 28, when record 25 was previously the last record in the database, will create records 26 and 27 as well as 28. Attempts to retrieve records that were created in this manner will result in an error return of DB_KEYEMPTY.

If a created record is not at the end of the database, all records following the new record will be automatically renumbered upward by 1. For example, the creation of a new record numbered 8 causes records numbered 8 and greater to be renumbered upward by 1. If a cursor was positioned to record number 8 or greater before the insertion, it will be shifted upward 1 logical record, continuing to reference the same record as it did before.

For these reasons, concurrent access to a recno database with the DB_RENUMBER flag specified may be largely meaningless, although it is supported.

DB_SNAPSHOT
This flag specifies that any specified *re_source* file be read in its entirety when *Db::open* is called. If this flag is not specified, the *re_source* file may be read lazily.

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)