NAME

nis_db, db_initialize, db_create_table, db_destroy_table, db_first_entry, db_next_entry, db_reset_next_entry, db_list_entries, db_remove_entry, db_add_entry, db_table_exists, db_unload_table, db_checkpoint, db_standby, db_free_result – NIS+ Database access functions

SYNOPSIS

cc [*flag* . . .] *file*. . . **–lnisdb –lnsl** [*library*. . .]

#include <rpcsvc/nis.h>
#include <rpcsvc/nis_db.h>

bool db_initialize(const char **dictionary_pathname***);**

db_status db_create_table(const char **table_name***, const table_obj** **table***);**

db_status db_destroy_table(const char **table_name***);**

db_result *db_first_entry(const char **table_name***, const int** *numattrs***,**
const nis_attr **attrs***);**

db_result *db_next_entry(const char **table_name***, const db_next_desc** **next_handle***);**

db_result *db_reset_next_entry(const char **table_name***,**
const db_next_desc **next_handle***);**

db_result *db_list_entries(const char **table_name***, const int** *numattrs***,**
const nis_attr **attrs***);**

db_result *db_remove_entry(const char **table_name***, const int** *numattrs***,**
const nis_attr **attrs***);**

db_result *db_add_entry(const char **table_name***, const int** *numattrs***,**
const nis_attr **attrs***, const entry_obj** **entry***);**

db_status db_table_exists(const char **table_name***);**

db_status db_unload_table(const char **table_name***);**

db_status db_checkpoint(const char **table_name***);**

db_status db_standby(const char **table_name* **);**

void db_free_result(db_result *);

MT-LEVEL

Unsafe

DESCRIPTION

These functions describe the interface between the NIS+ server and the underlying database. They are defined in the shared library **/usr/lib/libnisdb.so**.

The interface is a simple subset of a complete relational database and provides just those items that are needed by the NIS+ server daemon. When you replace the database, your interface routines should match these exactly. Also note that the database is responsible for verifying that the objects passed do not exceed the internal limits of the database being used.

The database's performance will directly affect the performance of the server. The default information base that is provided with NIS+ is the Structured Storage Manager (SSM). This is a memory based database that has been tuned for NIS+.

These routines should not be invoked by any NIS+ client. NIS+ clients should use the NIS+ tables API described in **nis_tables**(3N).

These routines only use the **table_obj**, **entry_obj** and the **nis_attr** structures defined in **<rpcsvc/nis.h>**. The NIS+ directory is itself stored in a table by the service daemon. This table has two columns, one searchable with the name of the object in it, the other non-searchable with binary XDRed data in it. The NIS+ server converts directory lookup requests in the namespace into table searches. The table it searches in response to these requests will have the same name as the directory of the name it is searching for.

The structure returned by the DB access routines is defined as:

```
enum db_status {DB_SUCCESS, DB_NOTFOUND, DB_NOTUNIQUE, DB_BADTABLE,
    DB_BADQUERY, DB_BADOBJECT, DB_MEMORY_LIMIT, DB_STORAGE_LIMIT,
    DB_INTERNAL_ERROR };

struct db_result {
        db_status       status;                         /* Result status */
        db_next_desc    nextinfo;                       /* descriptor */
        struct {
                        u_int       objects_len;
                        entry_obj   *objects_val;
        } objects;                                      /* A variable list
                                                        of objects */
        long            ticks;                          /* execution time in
                                                        microseconds */
};
```

For a complete description of NIS+ objects, see **nis_objects**(3N).

The structure **db_next_desc** should be used as an opaque handle for **db_next_entry( )** and **db_reset_next_entry( )**.

The **nis_attr** structure used in **db_first_entry** and other related functions is defined as follows:

```
struct nis_attr {
        char    *zattr_ndx;
        struct {
                u_int   zattr_val_len;
                char    *zattr_val_val;
        } zattr_val;
};
```

zattr_ndx is the name of the attribute. **zattr_val_len** is the value of the attribute **zattr_val_val**.

In **db_result**, the *objects* array contains objects if and only if the result returned in the *status* variable is **DB_SUCCESS**. A null pointer, instead of a pointer to a **db_result** structure, is returned if there is insufficient memory to create the structure.

db_initialize( ) is called prior to any interaction with the database. It takes as argument the pathname of the file that contains, or will contain, catalog information associated with the database.

db_create_table( ) creates a new table using the given table name and the table object. It returns TRUE if the table was successfully created; FALSE otherwise.

db_destroy_table( ) destroys the table of the given name. It returns TRUE if the destruction was successful; FALSE otherwise.

db_first_entry( ) returns a copy of the first entry in the specified table that satisfies the given attributes. If no attributes are supplied, a copy of the first entry in the table is returned. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements. The returned structure, **db_result**, contains a structure, **db_next_desc**, to be used as an argument to **db_next_entry( )** or **db_reset_next_entry( )**. **db_next_desc** should only be used only as an opaque handle. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_next_entry( ) returns a copy of the next entry as indicated by the *next_handle*. An initial call to **db_first_entry( )**, followed by a sequence of calls to **db_next_entry( )**, can be used to successfully obtain entries of an entire table or entries that satisfy the attributes supplied to **db_first_entry( )**. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_reset_next_entry( ) terminates the **db_first_entry( )/db_next_entry( )** sequence as indicated by *next_handle*, freeing any resources that have been used to maintain the sequence. After a call to **db_reset_next_entry( )**, a call to **db_next_entry( )** using the same *next_handle* would fail, returning a **DB_BADQUERY** reply. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_list_entries( ) returns copies of entries that satisfy the given attributes. **db_free_result( )** can be used to free up the returned **db_result** structure. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements.

db_remove_entry( ) removes all entries that satisfy the given attributes. **db_free_result( )** can be used to free up the returned **db_result** structure. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements.

db_add_entry( ) adds a copy of the given object to the specified table, replacing the one identified by the given attributes. If the given attributes identify more than one object, **DB_NOTUNIQUE** is returned. If no object is identified by the given attributes, the object is added. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_table_exists( ) provides an efficient way for the NIS+ service to detect that a table exists. This increases response time to the client and lowers the load on the server.

db_unload_table( ) is used by the service to unload or deactivate tables that are not currently being used. The service internally keeps track of access patterns to tables and will unload those tables that have not been accessed for a while. By unloading infrequently accessed tables, the service can minimize the amount of system resources for efficient operation.

db_checkpoint( ) organizes the contents of the table in a more efficient manner. Check-pointing may mean different things to different types of databases. It does not affect the logical contents of the table — operations and queries should return the same result before and after a checkpoint. For example, in a log-based system, checkpointing may mean incorporating log entries of updates accumulated since the previous checkpoint into the table.

db_free_result( ) frees up the space allocated by various functions listed on this manual page that return a **db_result** structure.

db_standby( ) is an advisory call to the database manager. This call informs the database that activity has slowed down and it can free up unnecessary resources such as file descriptors.

PROGRAMMING

Most of the routines in this library use an NIS+ *name* to identify the object that the user desires. The name must be in canonical form before being passed to the database because one server may be serving several namespaces and discrimination of the requested objects is accomplished by comparing the domain names.

DIAGNOSTICS

DB_SUCCESS The query or operation completed successfully and returned status.

DB_NOTFOUND
The name or entry that was named in the argument did not exist.

DB_NOTUNIQUE
An attempt was made to remove an entry from a table that is not uniquely specified.

DB_BADQUERY
The query that was submitted to the database was invalid (for example, it might name some nonexistent fields).

DB_BADTABLE
The table was corrupted.

DB_BADOBJECT
The fields of the object does not conform to the fields of the table to which it is being added.

DB_MEMORY_LIMIT
There is insufficient memory to complete the operation requested.

DB_STORAGE_LIMIT
There is insufficient file storage available to complete the operation requested.

DB_INTERNAL_ERROR
An internal error was encountered during the execution of the operation requested (either a programming error or an unrecoverable exception).

SEE ALSO

rpc.nisd(1M), **nis_objects**(3N), **nisfiles**(4)