RBerkeley: Getting Started with DB and R

Jeffrey A. Ryan

July 23, 2009

Contents

1 Overview

Berkeley DB[2] is an embedded database application distributed by Oracle for use in open source and commercial applications.

Widely deployed, Berkeley DB is used behind the scenes in many of the most popular applications and services in the world. It is designed to be fast, memory-efficient, and fully ACID compliant. Berkeley DB, also known commonly as DB, is shipped with APIs that allow for programatic access to its features. These directly supported language interfaces include C, C++, Java and Tcl.

Contributed bindings exists in a variety of programming languages, including Perl, Python, and Ruby.

RBerkeley adds an `R` [1] language binding to this pool of options.

This document is not to extoll the virtues of DB, but rather to provide a quick start to using the `R` interface. The general design philosophy of RBerkeley will be explained, and some small examples of using it in practice will be worked.

At present a large part of the very large API has been incorporated into RBerkeley, including support for most database functions, cursors, environments and mutexes. Not yet supported, though in development, are transactions, locks, and the rest of the API.

2 Design Philosophy

The RBerkeley interface is designed to be as close to the native C API as possible. Naming conventions are maintained, with some simple substitutions to allow for legal `R` function names. Most of the functionality exposed at the R level is simply a thin wrapper to an internal call to the C wrapper around the C API calls.

Some additional functionality and function calls are included to offer a more seamless user experience, as well as provide a more direct connection to the syntax of the C API, so as the official Oracle documentation can be used once a basic syntax conversion is performed.

3 Getting DB and RBerkeley

Berkeley DB is most likely already available on your system if you are running a unix or linux variant. The location of the library files are system dependant, and at present the configure script included with RBerkeley tries to identify the correct locations.

Installing DB

It is relatively simple to install the newest version of DB directly from Oracle by following the instructions included in the download from Oracle. The remainder of this section will assume the default location on POSIX systems. You may need to alter your configuration files to build RBerkeley.

Installing RBerkeley

Installing RBerkeley follows the standard R installation convention of:

```
R CMD INSTALL RBerkeley_0.0-1.tar.gz
```

[Additional details coming...]

4 Database Basics

The primary purpose of this document is not to show how to use DB, that is best understood by reading the official Oracle documentation. Our intention is instead to show basic usage patterns, from the R environment, using the RBerkeley package interface.

To begin, we will create a new database, add some R objects to it and read them back. This will give the most simplistic use case.

4.1 Opening a Database

Here we create a new database handle, and open our test database.

```
> dbh <- db_create()
> dbh
```

```
<DB Handle>
```

`db_create` can take some additional parameters, but we use the defaults here. The object returned is of class `DB`.

Next we open the database:

```
> ret <- db_open(dbh, txnid = NULL, file = "myDB.db", type = "BTREE",
+     flags = mkFlags(DB_CREATE, DB_EXCL))
> db_strerror(ret)
```

```
[1] "Successful return: 0"
```

The `db_open` call opens the underlying database. As with db_create, different parameters may be passed in. Some of these options will be covered in later documentation, but for now the defaults are sufficient. Users are directed to the official API[3] for usage details.

The return value from the `db_open` call is an integer value. This is the behavior of the underlying API. The utility function `db_strerror` can take this value and return a message to R. Additional error and message tools are available in the API, though currently disabled.

The `dbh` argument is the database handle (a **DB** struct in the official API). This is from the previous `db_create` call.

The `txnid` is the transaction handle (a **DB_TXN** struct in the official API). This may be specified as NULL if no transaction capability is needed, and in the current version of RBerkeley transaction support is disabled, so this needs to be passed as `txnid=NULL`.

The `file` argument is simply the name of the database we wish to open.

The `type` can be one of any supported DB access methods. These include BTREE, HASH, RECNO, QUEUE or UNKNOWN, the latter for opening a

database of unknown type. The `db_open` function will accept a variety of variations on these names, see that function for details.

The `flags` argument in this example is used to create the underlying database file if needed, i.e. if one is not present, [**DB_CREATE**], but *if and only if* one is not present [**DB_EXCL**]. Flags are central to many of the advanced features in DB, and it is crucial to understand what you can and can't specify. Once again, the main API documentation should be the considered the definitive guide.

One interesting non-API function found in RBerkeley is `mkFlags`; this function provides access to the internal constants defined by DB for use as flag parameters to be passed to many of the internal functions.

The C API allows for flags to be constructed via bitwise OR operations on predefined DB constants. In order to not map all constants into R, which would be tedious, error-prone, and difficult to maintain, the RBerkeley package passes a list of quoted or unquoted names from mkFlags into C code which in turn performs this bitwise operation. `mkFlags` thus allows for syntax very close in look and feel to the base C API, and at the same time makes for a robust and safe mechanism to pass arbitrary combinations of flags correctly.

```
> mkFlags(DB_CREATE)

[1] 1

> mkFlags(DB_EXCL)

[1] 1024

> mkFlags(DB_CREATE,DB_EXCL)

[1] 1025

> mkFlags(DB_CREATE,DB_EXCL,DB_EXCL) # bitwise OR duplicates: no change

[1] 1025
```

A programming note about this interface: unquoted symbols are converted to character vectors and in turn concatenated into one vector that is passed into the C level mkFlags function. If illegal values are passed they will simply be ignored in the calculation, and will produce a visible warning regarding the specific invalid flag or flags that failed to be processed. The return value will be unaffected by improper or duplicate settings.

4.2 Adding Records

A database needs to have content, if it is to be of use. Berkeley DB stores records in databases as simple key-value pairs. These records can be organized with a variety of access schemes, all of which are discussed at length in the official documentation.

The central difference between records in DB and records in a typical relational database is that DB has no notion of type. Records, both key and data are simply byte-strings. This provides tremendous flexibility to the programmer, as it enables data to be stored in the most natural format possible, or in a manner that is most in accordance with expected usage patterns.

Byte-strings make it quite easy to store native R objects as *key* or *data*, making additional processing unecessary. Of course this also leaves open the possibility of creating a preprocessing model to match any arbitrary schema that is needed by the final application.

The primary difference in the RBerkeley implementation of the DB API is that most R objects are passed to R's `serialize` before being sent to the database. At the DB level, the API simply takes whatever raw data is passed in without effort to process. The only exception is for the R object type `raw` which is passed in *a*s is. This convention allows for simplicity of use from within R, yet offers the benefit of providing a direct interface to the underlying flexibility of DB if the application/programmer demands it. All DB data must be be serialized, so this conversion from R objects (always represented as type RAW internally by RBerkeley) to bytes is carried out in the package's C layer.

To add a record, the package and API support two primary functions: `db_put` and `dbcursor_put`. These correspond to the official API functions `DB->put` and `DBcursor->put`, respectively. We'll take a look at each individually, as well as introduce the concept of the DB cursor.

4.2.1 db_put *aka* DB->put

The most basic way to add data into a database is with `db_put`. This takes a handful of arguments and adds a new record into the database, and returns as is usual for most of the API, an integer value of the success or failure.

An example or two is the best way to understand how to use.

```
> db_put(dbh, key = "Ross", data = "Ihaka")
> db_put(dbh, key = "Robert", data = "Gentleman")
```

This takes the two R character vectors, and adds them as a key and data into our database referenced by the `dbh` handle we created earlier.

Internally it should be noted that it is the actual object being stored, after being run through `serialize`, and not the characters themselves. If one wanted to simply store the raw character values, instead of R objects, he could convert to a raw vector before passing into db_put.

```
> db_put(dbh, key = charToRaw("Ross"), data = charToRaw("Ihaka"))
> charToRaw("Ihaka")

[1] 49 68 61 6b 61
```

4.2.2 Cursors

A second slightly more abstract way of adding keys is to use a cursor. A cursor in database terminology is really nothing more than a pointer to a record. A cursor can traverse a database, and provides a host of retrieval options that simply using `db_put` would be impossible for.

Using cursors or even explaining much beyond the above is outside the scope of this document. The preceeding description should be sufficient to have a cursory understanding what is happening internally. Yes, cursory.

To add a record with a cursor, we first need a valid cursor handle to our database. This is accomplished with a call to `db_cursor` using a valid `dbh` handle to an open database.

```
> dbc <- db_cursor(dbh)
```

As with most API calls, there are numerous argument that may be passed into the creation call. The cursor (mapping to the C API struct **DBC**) is now ready for use.

To put a record in the database, we now use the appropriate *c*ursor method:

```
> dbcursor_put(dbc, key = 100L, data = 5L, flags = mkFlags(DB_KEYLAST))

[1] 0
```

Again, the return value of 0 is an indicator of success.

Some important points regarding the flags allowed need to be understood. It is imperative to understand the underlying DB functionality before using cursors.

4.3 Retrieving Records

As with putting records in a database, we can retrieve records through two different mechanisms as well. The is the standard database retrieval method for DB that is via the DB->get method, available in RBerkeley via `db_get`. The second method works with cursors, as we had seen before. As before these methods may be used together, or exclusively.

4.3.1 db_get *aka* DB->get

The 'standard' way to fetch records from a database would be with the `db_get` method. This only requires a open database, and a functioning database handle (internally a pointer to a struct **DB**, for those following allong with the official documentation).

Depending on the flags specified to the db_get function it is possible to perform more advanced operations than the following examples will cover. By default, `flags=0L` for all calls not specifying a flag argument.

In the present version of RBerkeley, transaction support is disabled, and must be set to NULL. If not specified, this is the default.

```
> db_get(dbh, key = 100L)

 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 0d 00 00 00 01 00 00 00
[26] 05
```

As only raw values are stored by DB and RBerkeley, it is up to the calling code to interpret the resultant output. By default, the original *put* calls serialize the R objects. There is no default behavior for *get*. If the object was serialized with R's `serialize` function (the default) simply wrapping `db_get` with `unserialize` will return the original object.

```
> unserialize(db_get(dbh, key = 100L))

[1] 5
```

Only the data value associated with the given key is returned. To access data that is stored under duplicate (identical) keys, or perform more advanced query operations including partial matching, it is necessary to use cursors.

4.3.2 Cursors

Cursor *get* functions are similar to the simpler database get functions, in that a data value is returned for a given key.

Cursors can be far more flexible if need be. For instance, it is possible to iterate over all key/data pairs in a given database, simply by passing `NULL` for the key and data arguments. Flags are once again critical to the behavior of the queries.

The transaction support (currently disabled in RBerkeley) is set at the instantiation of the cursor, and therefore there is no `txnid` argument to cursor *get* calls. A few example to illustrate some behavior:

```
> res <- dbcursor_get(dbc, n = 1)
> res

[[1]]
[[1]]$key
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 04 52 6f 73 73

[[1]]$data
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 05 49 68 61 6b 61
```

This retrieves the current value at the current cursor position. One notable difference with cursor calls versus standard `db_get` is that a list of key/data results are returned. Each element of the list is a list containing an element named 'key' with the value of the key, and an element names 'data' with the value of the data. Multiple elements would produce a list of length n, where n

would be the lesser of the 'n' value specified, or the number of records returned by the query.

Another item of note is that the returned values contained as elements of the list are still in `RAW` form. Unserializing or otherwise converting into R objects would be the final step in most applications.

```
> lapply(res[[1]], unserialize)

$key
[1] "Ross"

$data
[1] "Ihaka"
```

It is also possible to find a specific record by using the DB_SET flag and specifying a key. This is similar to the traditional `db_get` results, though with the key returned as well.

```
> dbcursor_get(dbc, key = "Ross", flags = mkFlags("DB_SET"))

[[1]]
[[1]]$key
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 04 52 6f 73 73

[[1]]$data
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 05 49 68 61 6b 61
```

The `data` argument to the `dbcursor_get` function may be used to further specify a query. The main Oracle documentation should be referenced here, but a few examples will once again be illustrative.

```
> res <- dbcursor_get(dbc, key = "Ross", data = "Brawn", flags = mkFlags("DB_SET"))
> lapply(res[[1]], unserialize)

$key
[1] "Ross"

$data
[1] "Ihaka"
```

The above call, as you will note, returns something other than the data we were requesting. This is a result of the `flags` argument being set to *DB_SET*. DB simply returns key/data for the first element to match the key.

To prevent this behavior, set `flags=mkFlags("DB_GET_BOTH")`.

```
> dbcursor_get(dbc, key = "Ross", data = "Braun", flags = mkFlags("DB_GET_BOTH"))
```

```
[[1]]
NULL
```

Cursors can also be useful for iterating over a database's key/data records. To iterate over the entire database, it is necessary to have a new (uninitialized) cursor. Closing and re-opening the original cursor object would be just as good as creating a new one in most cases.

```
> dbcursor_close(dbc)

[1] 0

> dbc <- db_cursor(dbh)
> res <- dbcursor_get(dbc, flags = mkFlags("DB_NEXT"), n = 100)
> res

[[1]]
[[1]]$key
[1] 52 6f 73 73

[[1]]$data
[1] 49 68 61 6b 61


[[2]]
[[2]]$key
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 0d 00 00 00 01 00 00 00
[26] 64

[[2]]$data
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 0d 00 00 00 01 00 00 00
[26] 05


[[3]]
[[3]]$key
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 04 52 6f 73 73

[[3]]$data
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 05 49 68 61 6b 61


[[4]]
[[4]]$key
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
```

```
[26] 09 00 00 00 06 52 6f 62 65 72 74

[[4]]$data
 [1] 58 0a 00 00 00 02 00 02 09 00 00 02 03 00 00 00 00 10 00 00 00 01 00 00 00
[26] 09 00 00 00 09 47 65 6e 74 6c 65 6d 61 6e
```

Given the output returns, and with knowledge of the data contained, we can use the R language to easily find the keys in our database. We'll exclude the first record, as we didn't use serialize in that entry. Just like any storage, knowledge of your data is critical to proper processing.

```
> sapply(res[-1], function(x) unserialize(x$key))

[1] "100"    "Ross"   "Robert"
```

4.4 Removing Records

Sometimes it may be necessary to remove records from the database. As with *put* and *get* functionality, we can delete records using standard **DB** methods or use cursors. A few example on how this is done;

4.4.1 db_del *aka* DB->del

The basic DB method removes all records (duplicates included) matching the `key` argument. The function is quite straightforward. The `flags` argument, currently unused, must be set to zero or remain unspecified. The function returns the standard DB error values as integers.

As an example, this will check to see if a key exists before we delete it, delete it, then check again.

```
> db_strerror(db_exists(dbh, key = charToRaw("Ross")))

[1] "Successful return: 0"

> db_del(dbh, key = charToRaw("Ross"))

[1] 0

> db_strerror(db_exists(dbh, key = charToRaw("Ross")))

[1] "DB_NOTFOUND: No matching key/data pair found"
```

4.4.2 Cursors

An alternative approach, though a bit less obvious, is to use cursors. Cursor deletes will simply delete the record at the cursor's current position. There is no key argument, and the flags argument must be set to zero, per the DB API.

To delete the first record in the database, position the cursor using `dbcursor_get` with the flags argument set to "DB_FIRST". Then call the cursor delete function.

```
> firstrecord <- dbcursor_get(dbc, flags = mkFlags("DB_FIRST"))[[1]]
> db_strerror(dbcursor_del(dbc))

[1] "Successful return: 0"

> dbcursor_get(dbc, key = firstrecord$key, flags = mkFlags("DB_SET"))

[[1]]
NULL
```

4.5 Closing a Database

To ensure data integrity, in a persistant state, it is required to close a database and open cursors before exiting a session.

A database needs to have no open cursors in it in order to be closed, so it is first important to close any outstanding cursors we have created. After that, we may simply call `db_close` on the open database handle. A warning about being unable to use either cursor or database handle will be issued once the method is called.

```
> dbcursor_close(dbc)

[1] 0

> db_close(dbh)

[1] 0
```

5 Conclusion

RBerkeley provides low-level API access to the Berkeley DB embedded database library. While this document merely touched on basic usage, a full suite of API functionality is available to the R user via the 70+ functions currently shipped with the RBerkeley package.

Testing, documentation, and further methods will be added in upcoming RBerkeley version.

References

[1] R Development Core Team: *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org

[2] Oracle Berkeley DB 4.7.25: http://www.oracle.com/technology/documentation/berkeley-db/db/index.html

[3] Oracle Berkeley DB C API: http://www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html