ODBC Connectivity

by Brian Ripley

Department of Statistics, University of Oxford

ripley@stats.ox.ac.uk

February 26, 2015

Package **RODBC** implements ODBC database connectivity. It was originally written by Michael Lapsley (St George's Medical School, University of London) in the early days of R (1999), but after he disappeared in 2002, it was rescued and since much extended by Brian Ripley. Version 1.0-1 was released in January 2003, and **RODBC** is nowadays a mature and much-used platform for interfacing R to database systems.

Thanks to Marc Schwartz for contributing some of the experiences here. See also the archives of the `R-sig-db` mailing list.

1   ODBC Concepts

ODBC aims to provide a common API for access to SQL[1]-based database management systems (DBMSs) such as MySQL[2], PostgreSQL, Microsoft Access and SQL Server, DB2, Oracle and SQLite. It originated on Windows in the early 1990s, but ODBC *driver managers* `unixODBC` and `iODBC` are nowadays available on a wide range of platforms (and `iODBC` shipped with versions of OS X prior to 10.9). The connection to the particular DBMS needs an *ODBC driver*: these may come with the DBMS or the ODBC driver manager or be provided separately by the DBMS developers, and there are third-party[3] developers such as Actual Technologies, Easysoft and OpenLink. (This means that for some DBMSs there are several different ODBC drivers available, and they can behave differently.)

Microsoft provides drivers on Windows for non-SQL database systems such as DBase and FoxPro, and even for flat files and Excel spreadsheets. Actual Technologies sell a driver for OS X that covers (some) Excel spreadsheets and flat files.

A connection to a specific database is called a *Data Source Name* or DSN (see http://en.wikipedia.org/wiki/Database_Source_Name). See Appendix B for how to set up DSNs on your system. One of the greatest advantages of ODBC is that it is a cross-platform client-server design, so it is common to run R on a personal computer and access

---

[1]SQL is a language for querying and managing data in databases—see http://en.wikipedia.org/wiki/SQL.

[2]and its fork, MariaDB

[3]but there are close links between `unixODBC` and Easysoft, and `iODBC` and OpenLink.

data on a remote server whose OS may not even be known to the end user. This does rely on suitable ODBC drivers being available on the client: they are for the major cross-platform DBMSs, and some vendors provide 'bridge' drivers, so that for example a 'bridge' ODBC driver is run on a Linux client and talks to the Access ODBC driver on a remote Windows machine.

ODBC provides an abstraction that papers over many of the differences between DBMSs. That abstraction has developed over the years, and **RODBC** works with ODBC version 3. This number describes both the API (most drivers nowadays work with API 3.51 or 3.52) and capabilities. The latter allow ODBC drivers to implement newer features partially or not at all, so some drivers are much more capable than others: in the main **RODBC** works with basic features. ODBC is a superset of the ISO/IEC 9075-3:1995 SQL/CLI standard.

A somewhat biased overview of ODBC on Unix-alikes can be found at http://www.easysoft.com/developer/interfaces/odbc/linux.html.

2 Basic Usage

Two groups of functions are provided in **RODBC**. The mainly internal `odbc*` commands implement low-level access to C-level ODBC functions with similar[4] names. The `sql*` functions operate at a higher level to read, save, copy and manipulate data between data frames and SQL tables. The two low-level functions which are commonly used make or break a connection.

2.1 Making a connection

ODBC works by setting up a *connection* or *channel* from the client (here **RODBC**) to the DBMSs as specified in the DSN. Such connections are normally used throughout a session, but should be closed explicitly at the end of the session—however **RODBC** will clear up after you if you forget (with a warning that might not be seen in a GUI environment). There can be many simultaneous connections.

The simplest way to make a connection is

```
library(RODBC)
ch <- odbcConnect("some_dsn")
```

and when you are done with it,

```
close(ch)
# or if you prefer
odbcClose(ch)
```

The connection object `ch` is how you specify one of potentially many open connections, and is the first argument to all other **RODBC** functions. If you forget the details, printing it will give some summary information.

---

[4]in most cases with prefix `SQL` replacing `odbc`.

If the DBMS user and password are needed and not stored in the DSN, they can be supplied by e.g.

```
ch <- odbcConnect("some_dsn", uid = "user", pwd = "****")
```

Users of the R GUI under Windows[5] have another possibility: if an incompletely specified DSN is given, the driver-specific Data Source dialog box will pop up to allow it to be completed.

More flexibility is available *via* function `odbcDriverConnect`, which works with a *connection string*. At its simplest it is

```
"DSN=dsn;UID=uid;PWD=pwd"
```

but it can be constructed without a DSN by specifying a driver directly *via* `DRIVER=`, and more (in some cases many more) driver-specific parameters can be given. See the documentation for the driver (and Appendix A) for more details.

2.2 Reading from a database

where 'database' can be interpreted very widely, including for example Excel spreadsheets and directories of flat files.

The simplest and most common use of **RODBC** is to extract data from databases held on central database servers. Such access is read-only, and this can be enforced by settings in the DSN or *via* permission settings (also known as *privileges*) on the database.

To find out what tables are accessible from a connection `ch`, use

```
sqlTables(ch)
```

Some drivers will return all visible table-like objects, not just those owned by you. In that case you may want to restrict the scope by e.g.

```
sqlTables(ch, tableType = "TABLE")
sqlTables(ch, schema = "some_pattern")
sqlTables(ch, tableName = "some_pattern")
```

The details are driver-specific but in most cases *some_pattern* can use wildcards[6] with *underscore* matching a single character and *percent* matching zero or more characters. Since underscore is a valid character in a table name it can be handled literally by preceding it by a backslash—but it is rarely necessary to do so.

A table can be retrieved as a data frame by

```
res <- sqlFetch(ch, "table_name")
```

If it has many rows it can be retrieved in sections by

```
res <- sqlFetch(ch, "table_name", max = m)
res <- sqlFetchMore(ch, "table_name", max = m)
...
```

---

[5]This does not work from `Rterm.exe`.

[6]these are the SQL wildcards used for example in `LIKE` clauses.

It is often necessary to reduce the data to be transferred: we have seen how to subset rows, but it can be more effective to restrict the columns or to return only rows meeting some conditions. To find out what columns are available, use `sqlColumns`, for example

```
> sqlColumns(ch, "USArrests")
  TABLE_CAT TABLE_SCHEM TABLE_NAME COLUMN_NAME DATA_TYPE TYPE_NAME COLUMN_SIZE
1   ripley       <NA>   USArrests       State         12   varchar         255
2   ripley       <NA>   USArrests      Murder          8    double          15
3   ripley       <NA>   USArrests     Assault          4   integer          10
4   ripley       <NA>   USArrests    UrbanPop          4   integer          10
5   ripley       <NA>   USArrests        Rape          8    double          15
...
```

Then an *SQL Query* can be used to return part of the table, for example (MySQL on Linux)

```
> sqlQuery(sh, paste("SELECT State, Murder FROM USArrests",
+                     "WHERE Rape > 30 ORDER BY Murder"))
        State Murder
1    Colorado    7.9
2     Arizona    8.1
3  California    9.0
4      Alaska   10.0
5  New Mexico   11.4
6    Michigan   12.1
7      Nevada   12.2
8     Florida   15.4
```

Note that although there are standards for SQL, all the major producers of DBMSs have their own dialects, so for example on the Oracle and DB2 systems we tested this query had to be given as

```
> sqlQuery(ch, paste('SELECT "State", "Murder" FROM "USArrests"',
+                     'WHERE "Rape" > 30 ORDER BY "Murder"'))
```

or even in upper case. Describing how to extract data from databases is the *forte* of the SQL language, and doing so efficiently is the aim of many of the DBMSs, so this is a very powerful tool. To learn SQL it is best to find a tutorial specific to the dialect you will use; for example Chapter 3 of the MySQL manual is a tutorial. A basic tutorial which covers some common dialects[7] can be found at http://www.1keydata.com/sql/sql.html: tutorials on how to perform common tasks in several commonly used DBMSs are available at http://sqlzoo.net/.

2.3   Table Names

SQL-92 expects both table and column names to be alphanumeric plus underscore, and **RODBC** does not in general support vendor extensions (for example Access allows spaces). There are some system-specific quoting schemes: Access and Excel allow table names to be enclosed in [ ] in SQL queries, MySQL (by default) quotes *via* backticks, and most other systems use the ANSI SQL standard of double quotes.

---

[7]MySQL, Oracle and Microsoft SQL Server.

The `odbcConnnect` function allows the specification of the quoting rules for names **RODBC** itself sends, but sensible defaults[8] are selected. Users do need to be aware of the quoting issue when writing queries for `sqlQuery` themselves.

Note the underscore is a wildcard character in table names for some of the functions, and so may need to be escaped (by backslash) at times.

Normally table names containing a period are interpreted as references to another schema (see below): this can be suppressed by opening the connection with argument `interpretDot = FALSE`.

2.4   Types of table

The details are somewhat DBMS-specific, but 'tables' usually means 'tables, views or similar objects'.

In some systems 'tables' are physical objects (files) that actually store data—Mimer calls these *base tables*. For these other 'tables' can be derived that present information to the user, usually called 'views'. The principal distinctions between a (base) table and a view are

- Using `DROP` on a table removes the data, whereas using it on a view merely removes the convenient access to a representation of the data.

- The access permission (*privilege*) of a view can be very different from those of a table: this is commonly used to hide sensitive information.

A view can contain a subset of the information available in a single table or combine information from two or more tables.

Further, some DBMSs distinguish between tables and views generated by ordinary users and *system tables* used by the DBMS itself. Where present, this distinction is reflected in the result of `sqlTable()` calls.

Some DBMSs support *synonyms* and/or *aliases* which are simply alternative names for an existing table/view/synonym, often those in other schemas (see below).

Typically tables, views, synonyms and aliases share a name space and so must have a name that is unique (in the enclosing schema where schemas are implemented).

3   Writing to a Database

To create or update a table in a database some more details need to be considered. For some systems, all table and column names need to be lower case (e.g. PostgreSQL, MySQL on Windows) or upper case (e.g. some versions of Oracle). To make this a little easier, the

---

[8]backticks for MySQL, `[ ]` for the Access and Excel convenience wrappers, otherwise ANSI double quotes.

`odbcConnect` function allows a remapping of table names to be specified, and this happens by default for DBMSs where remapping is known to be needed.

The main tool to create a table is `sqlSave`. It is safest to use this after having removed any existing table of the same name, which can be done by

`sqlDrop(ch, "`*table_name*`", errors = FALSE)`

Then in the simplest usage

`sqlSave(ch, `*some_data_frame*`)`

creates a new table whose name is the name of the data frame (remapped to upper or lower case as needed) and with first column `rownames` the row names of the data frame, and remaining columns the columns of the data frame (with names remapped as necessary). For the many options, see the help page.

`sqlSave` works well when asked to write integer, numeric and reasonable-length[9] character strings to the database. It needs some help with other types of columns in mapping to the DBMS-specific types of column. For some drivers it can do a good job with date and date-time columns; in others it needs some hints (and e.g. for Oracle dates are stored as date-times). The files in the `RODBC/tests` directory in the sources and the installed file `tests.R` provide some examples. One of the options is the `fast` argument: the default is `fast = TRUE` which transfers data in binary format: the alternative is `fast = FALSE` which transfer data as character strings a row at a time—this is slower but can work better with some drivers (and worse with others).

The other main tool for writing is `sqlUpdate` which is used to change rows in an existing table. Note that **RODBC** only does this in a simple fashion, and on up-market DBMSs it may be better to set cursors and use direct SQL queries, or at least to control transactions by calls to `odbcSetAutoCommit` and `odbcEndTran`. The basic operation of `sqlUpdate` is to take a data frame with the same column names (up to remapping) as some or all of the columns of an existing table: the values in the data frame are then used either to replace entries or to create new rows in the table.

Rows in a DBMS table are in principle unordered and so cannot be referred to by number: the sometimes tricky question is to know what rows are to replaced. We can help the process by giving one or more `index` columns whose values must match: for a data frame the row names are often a good choice. If no `index` argument is supplied, a suitable set of columns is chosen based on the properties of the table.

3.1 Primary keys and indices

When a table is created (or afterwards) it can be given additional information to enable it to be used effectively or efficiently.

---

[9]which of course depends on the DBMS. Almost all have an implementation of `varchar` that allows up to 255 bytes or characters, and some have much larger limits. Calling `sqlTypeInfo` will tell you about the data type limits.

Primary keys are one (usually) or more columns that provide a reliable way to reference rows in the table: values of the primary key must be unique and not `NULL` (SQL parlance for 'missing'). Primary keys in one table are also used as *foreign keys* in another table: this ensure that e.g. values of `customer_id` only take values which are included in the primary key column of that name in table `customers`. Support of foreign keys is patchy: some DBMSs (e.g, MySQL prior to 6.0) accept specifications but ignore them.

RODBC allows primary keys to be set as part of the `sqlSave()` function when it creates a table: otherwise they can be set by `sqlQuery()` in DBMS-specific ways (usually by `ALTER TABLE`).

Columns in a table can be declared as `UNIQUE`: primary keys and such columns are usually used as the basis for table indices, but other indices (sometimes called *secondary indices*) can be declared by a `CREATE INDEX` SQL command. Whether adding primary keys or other indices has any effect on performance depends on the DBMS and the query.

4 Data types

This can be confusing: R has data types (including `character`, `double`, `integer` and various classes including `Date` and `POSIXct`), ODBC has both C and SQL data types, the SQL standards have data types and so do the various DBMSs *and they all have different names* and different usages of the same names.

Double- and single-precision numeric values and 32- and 16-bit integers (only) are transferred as binary values, and all other types as character strings. However, unless `as.is = TRUE`, `sqlGetResults` (used by all the higher-level functions to return a data frame) converts character data to a date/date-time class or *via* `type.convert`.

You can find out the DBMS names for the data types used in the columns of a table by a call to `sqlColumns`, and further information is given on those types in the result of `sqlTypeInfo`. For example in MySQL,

```
  TABLE_CAT TABLE_SCHEM TABLE_NAME COLUMN_NAME DATA_TYPE TYPE_NAME COLUMN_SIZE
1   ripley        <NA>   USArrests       State        12   varchar         255
2   ripley        <NA>   USArrests      Murder         8    double          15
3   ripley        <NA>   USArrests     Assault         4   integer          10
4   ripley        <NA>   USArrests    UrbanPop         4   integer          10
5   ripley        <NA>   USArrests        Rape         8    double          15
  BUFFER_LENGTH DECIMAL_DIGITS NUM_PREC_RADIX NULLABLE REMARKS COLUMN_DEF
1           255             NA             NA        0                 ''
2             8             NA             NA        1               <NA>
3             4              0             10        1               <NA>
4             4              0             10        1               <NA>
5             8             NA             NA        1               <NA>
  SQL_DATA_TYPE SQL_DATETIME_SUB CHAR_OCTET_LENGTH ORDINAL_POSITION IS_NULLABLE
1            12               NA               255                1          NO
2             8               NA                NA                2         YES
3             4               NA                NA                3         YES
4             4               NA                NA                4         YES
5             8               NA                NA                5         YES
```

This gives the DBMS data by name and by number (twice, once the number used in the DBMS and once that used by SQL—they agree here). Other things of interest here are

the column size, which gives the maximum size of the character representation, and the two columns about 'nullable' which indicate if the column is allowed to contain missing values (SQL NULLs).

The result of `sqlTypeInfo` has 19 columns and in the version of MySQL used here, 52 types. We show a small subset of the more common types:

```
> sqlTypeInfo(channel)[<...>, c(1:3,7,16)]
    TYPE_NAME DATA_TYPE COLUMN_SIZE NULLABLE SQL_DATATYPE
1         bit        -7           1        1           -7
2     tinyint        -6           3        1           -6
6      bigint        -5          19        1           -5
18       text        -1       65535        1           -1
19  mediumtext       -1    16777215        1           -1
20   longtext        -1  2147483647        1           -1
22       char         1         255        1            1
23    numeric         2          19        1            2
24    decimal         3          19        1            3
25    integer         4          10        1            4
37   smallint         5           5        1            5
41     double         6          15        1            6
43      float         7           7        1            7
45     double         8          15        1            8
47       date        91          10        1            9
48       time        92           8        1            9
49       year         5           4        1            5
50   datetime        93          21        1            9
51  timestamp        93          14        0            9
52    varchar        12         255        1           12
```

Note that there are both duplicate type names and duplicate type numbers.

Most DBMSs started with their own data types and later mapped the standard SQL data types on to them, although these may only be partially implemented. Some DBMSs allow user-defined data types, for example enumerations.

Commonly used data types fall into a number of groups:

Character types Character types can be classified three ways: fixed or variable length, by the maximum size and by the character set used. The most commonly used types[10] are `varchar` for short strings of variable length (up to some maximum) and `char` for short strings of fixed length (usually right-padded with spaces). The value of 'short' differs by DBMS and is at least 254, often a few thousand—often other types will be available for longer character strings. There is a sanity check which will allow only strings of up to 65535 bytes when reading: this can be removed by recompiling **RODBC**.

Many other DBMSs have separate types to hold Unicode character strings, often with names like `nvarchar` or `wvarchar`. Note that currently **RODBC** only uses the current locale for character data, which could be UTF-8 (and will be on OS X and in many cases on Linux and other Unix-alikes), but is never UCS-2 as used on Windows. So if character data is stored in the database in Unicode, it will be translated (with a possible loss of information) in non-Unicode locales. (This may change in future versions of **RODBC**.)

---

[10]the SQL names for these are `CHARACTER VARYING` and `CHARACTER`, but these are too cumbersome for routine use.

Some DBMSs such as PostgreSQL and Microsoft SQL Server allow variable-length character strings of length only limited by resources. These do not fit well with the ODBC model that requires buffers to be allocated to transfer character data, and so such types may be subjected (by the ODBC driver) to a fixed limit or not work at all.

Integer types Most DBMSs have types for 32-bit (`integer`, synomyn `int`) and 16-bit (`smallint`) integers. Some, including MySQL, also have unsigned versions and 1-bit, 8-bit and 64-bit integer types: these further types would usually be transferred as character strings and converted on reading to an `integer` or `double` vector.

Type names `int2`, `int4` and `int8` are common as synonyms for the basic type names.

The SQL standard does not require `integer` and `smallint` to be binary (rather than decimal) types, but they almost always are binary.

Note that 64-bit integers will be transferred as character strings and read by `sqlGetResults` as character vectors or (for $2^{31} \leq |x| < 2^{53}$) as `double` vectors.

Floating-point types The basic SQL floating-point types are 8 and 7 for double- and single-precision binary types. The SQL names are `double precision` and `real`, but beware of the variety of names. Type 6 is `float` in the standard, but is used by some DBMSs[11] for single-precision and by some for double-precision: the forms `float(24)` and `float(53)` are also commonly supported.

You should not assume that these types can store `Inf`, `-Inf` or `NaN`, but they often can.

Other numeric types It is common to store decimal quantities in databases (e.g. currency amounts) and types 2 and 3 are for decimals. Some DBMSs have specialized types to handle currencies, e.g. `money` in Microsoft SQL Server.

Decimal types have a *precision* (the maximum number of significant decimal digits) and *scale* (the position of the decimal point). `numeric` and `decimal` are usually synonymous, but the distinction in the standards is that for `numeric` the precision is exact whereas for `decimal` the DBMS can use a larger value than that specified.

Some DBMSs have a type `integer($p$)` to represent up to p decimal digits, and this may or may not be distinct from `decimal($p$, 0)`.

DBMSs do not necessarily fully implement decimal types, e.g. MySQL currently stores them in binary and used to store them as character strings.

Dates and times The handling of dates and times is very much specific to the DBMS. Some allow fractional seconds in date-times, and some do not; some store timezones with date-times or always use UTC and some do not, and so on. Usually there are also types for time intervals.

All such types are transferred as character strings in **RODBC**.

---

[11]In Oracle the `FLOAT` type is a decimal and not a binary type.

Binary types These are less common, and unsupported by **RODBC** prior to version 1.3-0. They parallel character types in that they are a sequence of bytes of fixed or variable length, sometimes with additional types for long sequences: there are separate ODBC types for `SQL_BINARY`, `SQL_VARBINARY` and `SQL_LONGVARBINARY`.

Binary types can currently only be read as such, and they are returned as column of class `"ODBC_binary"` which is a list of raw vectors.

It is possible (but rare) for the DBMS to support data types that the ODBC driver cannot handle.

4.1   Data types when saving a data frame

When `sqlSave` creates a table, there is some choice as to the SQL data types used.

The default is to select the SQL data type from the R type via the `typeInfo` argument to `sqlSave`. If this is not supplied (usual) a default mapping is looked up using `getSqlTypeInfo()` or by interrogating `sqlTypeInfo()`. This will almost always produce the correct mapping for numeric, integer and character columns of up to 254 characters (or bytes). In other cases (include dates and date-times) the desired SQL type can be specified for each column *via* the argument `varTypes`, a named character vector with names corresponding to (some of) the names in the data frame to be saved.

Only a very few DBMSs have a logical data type and the default mapping is to store R logical vectors as `varchar(5)`. For others DBMSs `BIT`, `TINYINT` or an enumeration type could be used (but the column may be need to be converted to and from a suitable representation). For example, in MySQL we could use `enum('FALSE', 'TRUE')`, but this is actually stored as `char(5)`. Note that to represent `NA` the SQL data type chosen needs to be nullable, which `BIT` often is not. (Mimer has a nullable data type `BOOLEAN` but this is not supported by the ODBC client.)

4.2   SQLite

SQLite's concept of 'data type' is anomalous: version 3 does recognize types of data (in version 2 everything was a character string), but it does not have a fixed type for a column in a table (although the type specified in the `CREATE TABLE` statement is a 'recommended' type for the values of that column). Every value is categorized as null, integer (of length 1, 2, 3, 4, 6 or 8 bytes), double, text (UTF-8 or UTF-16) or BLOB (a sequence of bytes). This does not fit well with the ODBC interface which pre-determines a type for each column before reading or writing it: the 'SQLite ODBC' driver falls back to a `SQL_VARCHAR` or `SQL_LONGVARCHAR` type if the column type is not available.

4.3   ODBC data types

ODBC defines two sets of data types: *SQL data types* and *C data types*. SQL data types indicate the data types of data stored at the data source using standard names. C data

```
SQL_CHAR             1    SQL_LONGVARCHAR     -1
SQL_NUMERIC          2    SQL_BINARY          -2
SQL_DECIMAL          3    SQL_VARBINARY       -3
SQL_INTEGER          4    SQL_LONGVARBINARY   -4
SQL_SMALLINT         5    SQL_BIGINT          -5
SQL_FLOAT            6    SQL_TINYINT         -6
SQL_REAL             7    SQL_BIT             -7
SQL_DOUBLE           8    SQL_WCHAR           -8
SQL_DATETIME         9    SQL_WVARCHAR        -9
SQL_INTERVAL        10    SQL_WLONGVARCHAR   -10
SQL_TIMESTAMP       11    SQL_GUID           -11
SQL_VARCHAR         12
SQL_TYPE_DATE       91
SQL_TYPE_TIME       92
SQL_TYPE_TIMESTAMP  93
```

Table 1: Mapping between ODBC SQL data type names and numbers. (GUIDs are 16-byte numbers, Microsoft's implementation of UUIDs.)

types indicate the data types used in the compiled code in the application (here **RODBC**) when transferring data and are the same for all drivers.

The ODBC SQL data types are abstractions of the data types discussed above with names like SQL_INTEGER. They include SQL_LONGVARCHAR for large character types and SQL_WVARCHAR for Unicode character types. It is usually these types that are returned (by number) in the SQL_DATA_TYPE column of the result of sqlColumns and SQL_DATATYPE column of the result of sqlTypeInfo. The mapping from names to numbers is given in table 1.

The only ODBC C data types currently used by **RODBC** are SQL_C_DOUBLE, SQL_C_SLONG (32-bit signed integers) and SQL_C_CHAR for reading and writing, and SQL_C_FLOAT (single-precision), SQL_C_SSHORT (16-bit signed integers) and SQL_C_BINARY for reading from the database.

http://msdn.microsoft.com/en-us/library/ms713607%28VS.85%29.aspx is the defintiive source of information about ODBC data types.

5   Schemas and Catalogs

This is a more technical section: few users will need to deal with these concepts.

'Schemas'[12] are collections of objects (such as tables and views) within a database that are supported by some DBMSs: often a separate schema is associated with each user (and 'schema' in ODBC 3 replaced 'owner' in ODBC 2). In SQL-92, schemas are collected in a 'catalog' which is often implemented as a database. Where schemas are implemented, there is a *current schema* used to find unqualified table names, and tables in other schemas can be referred to within SQL queries using the *schema.table* notation. You can think of

---

[12]which is the usual plural in this technical usage, athough *schemata* is more usual in English.

a schema as analogous to a name space; it allows related objects to be grouped together without worrying about name clashes with other groups. (Some DBMSs will search for unqualified table names in a search path: see the detailed descriptions below.)

Note that 'schema' is used in another sense in the database literature, for the design of a database and in particular of tables, views and privileges.

Here are some details of various DBMSs' interpretations of `catalog` and `schema` current at the time of writing (mid 2009). (These descriptions are simplistic, and in some cases experimental observations.)

- SQLite uses dotted names for alternative databases that are attached by an `ATTACH DATABASE` command.[13] There is a search path of databases, so it is only necessary to use the dotted name notation when there are tables of the same name on attached databases. The initial database is known as `main` and that used for temporary tables as `temp`.

- MySQL uses `catalog` to refer to a database. In MySQL's parlance, 'schema' is a little-used synonym for 'database'.

- PostgreSQL only allows a session to access one database, and does not use 'catalog' except to refer to the current database. Version 7.3 introduced schemas—users can create their own schemas with a `CREATE SCHEMA` query. Tables are by default in the `public` schema, and unqualified table names are searched for along a 'search path' of schemas (by default, containing `public`).

- Oracle uses schemas as synonymous with 'owner' (also known as 'user'). There is no way for a user to create additional schemas (that is not what `CREATE SCHEMA` does in Oracle).

- IBM DB2 uses schemas as name spaces for objects that may lie on different databases: using *aliases* allows objects to be in more than one schema. The initial current schema is named the same as the user (`SQLID` in DB2 parlance), but users can create additional schemas with `CREATE SCHEMA` statements.

- Microsoft SQL Server 2008 uses both `catalog` and `schema`, `catalog` for the database and `schema` for the type of object, e.g. `"sys"` for most of the system tables/views and (default) `"dbo"` for user tables. Further schemas can be created by users. The default schema for a user can be set when the user is created and changed *via* `ALTER USER`.

  Prior to SQL Server 2005, 'schema' meant 'user', and the search path for unqualified names was the database user then `"dbo"`.

- The Microsoft Excel and Access ODBC drivers do not use schemas, but do use `catalog` to refer to other database/spreadsheet files.

- Mimer (www.mimer.com) uses schemas which are normally the same as users (which it calls *IDENT*s), but users can create additional schemas with `CREATE SCHEMA` statements. There are also system schemas. Mimer uses 'schemata' as the plural of schema.

---

[13]and may be subsequently detached by a `DETACH DATABASE` command

It is often possible to use `sqlTables` to list the available catalogs or schemas: see its help page for the driver-specific details.

RODBC usually works with tables in the current schema, but unless the connection was opened with `interpretDot = FALSE` most functions will attempt to interpret the 'dotted name' notation. The interpretation depends on the DBMS: the SQL-92 meaning is *schema*.*table* and this is accepted by PostgreSQL, Microsoft SQL Server, Oracle, DB2 and Mimer. However, MySQL uses *database*.*table*, and the functions try[14] that interpretation if they recognize a MySQL driver. Some DBMSs allow more than two components, but these are not currently supported by the **RODBC** functions.

Functions `sqlTables`, `sqlColumns` and `sqlPrimaryKeys` have arguments `catalog` and `schema` which in principle allow tables in other schemas to be listed or examined: however these are only partially implemented in many current ODBC drivers. See the help page for `sqlTables` for some further details.

For other uses, the trick is to select the schema(s) you want to use, which is done *via* an SQL statement sent by `sqlQuery`. For Oracle you can set the default schema (owner) by

`ALTER SESSION SET CURRENT_SCHEMA =` *schema*

whereas for PostgreSQL the search path can be changed *via*

`SET search_path TO` *schema1*,*schema2*.

In DB2, creating an alias in the current schema can be used to access tables in other schemas, and a `CURRENT SCHEMA` query can be used to change the current schema. In MySQL and Microsoft SQL Server a database can be selected by a `USE` *database* query.

6   Internationalization Issues

Internationalization issues are made more complex by ODBC being a client-server system, and the ODBC client (**RODBC**) and the server may be running on different machines with different OSes on different continents. So the client may need some help.

In most cases numeric data are transferred to and from R in binary form, so the representation of the decimal point is not an issue. But in some cases it could be (e.g. decimal rather than binary SQL data types will be transferred as character strings) and then the decimal point to be used will be taken from `options("dec")`: if unset this is set when **RODBC** is loaded from the setting of the current locale on the machine running R (*via* `Sys.localeconv`). Some ODBC drivers (e.g. for Microsoft SQL Server, Oracle) allow the locale ('NLS') to be used for numeric values to be selected for the connection.

The other internationalization issue is the character encoding used. When R and the DBMS are running on the same machine this is unlikely to be an issue, and in many cases the ODBC driver has some options to translate character sets. SQL is an ANSI (US) standard, and DBMSs tended to assume that character data was ASCII or perhaps

---

[14]currerntly this is stymied by bugs in the ODBC driver, so `SQLColumns` is unable to report on tables in specified databases.

8-bit. More recently DBMSs have started to (optionally or by default) to store data in Unicode, which unfortunately means UCS-2 on Windows and UTF-8 elsewhere. So cross-OS solutions are not guaranteed to work, but most do.

Encoding issues are best resolved in the ODBC driver or in DBMS settings. In the unusual case that this cannot be done, the `DBMSencoding` argument to `odbcDriverConnect` allows for recoding when sending data to or from the ODBC driver and thence the DBMS.

7 Excel Drivers

The Microsoft Excel ODBC drivers (Windows only) have a number of peculiarities which mean that it should be used with care.

It seems that their concept of a 'table' is principally a *named range*. They treat worksheets as system tables, and append a dollar to their name (making then non-standard SQL table names: the quoting convention used is to enclose such names in square brackets).

Column names are taken as the first row of the named range/worksheet. Non-standard SQL names are allowed here too, but the driver maps `.` to `#` in column names. Annoyingly, `sqlTables` is allowed to select named ranges only by `tableType = "TABLE"` but not to select only worksheets.

There are at least two known problems with reading columns that do not have a format set *before* data entry, and so start with format 'General'. First, the driver uses the first few rows to determined the column type, and is over-fond of declaring 'Numeric' even when there are non-numeric entries. The default number of rows consulted is 8, but attempts to change this in the DSN setup are ignored. Second, if a column is declared as 'Text', numeric entries will be read as SQL nulls and hence R `NA`s. Unfortunately, in neither case does reformatting the column help.

The connection is by default read-only. It is possible to de-select this in the DSN (and the convenience wrapper `odbcConnectExcel` has a `readOnly = FALSE` argument to do so), but this does not support deletion, including SQL `DROP`, `DELETE`, `UPDATE` and `ALTER` statements). In particular, `sqlDrop` will remove the data in a worksheet but not the worksheet itself. The driver does allow a worksheet to be updated by `sqlUpdate`, and for a new worksheet (with a different name from existing worksheets) to be created by `sqlSave` (which also creates a named range).

As far as we know, no similar issues affect the Actual Technologies OS X Excel driver: however, it allows only read-only access to Excel files and does not support Excel 2007-and-later `.xlsx` files.

8 DBMS-specific tidbits

This section covers some useful DBMS-specific SQL commands and other useful details.

Recent versions of several DBMSs have a schema `INFORMATION_SCHEMA` that holds many predefined system views. These include MySQL (the name of a database, mainly populated beginning with MySQL 5.1), Microsoft SQL Server and Mimer.

MySQL

Comments about MySQL are mostly applicable to its forks such as MariaDB.

We have already mentioned `USE` *database* as the way to change the database in use. `SHOW DATABASES` lists the databases 'for which you have some kind of privilege', and can have a `LIKE` clause to restrict the result to some pattern of database names.

The `DESCRIBE` *table* command is a compact way to get a description of a table or view, similar to the most useful parts of the result of a call to `sqlColumns`. (It is also known as `SHOW COLUMNS FROM` *table*.)

`SHOW TABLES` is the command to produce a table of the tables/views on the current database, similar to `sqlTables`. For example,

```
> sqlQuery(channel, "USE ripley")
[1] "No Data"
> sqlQuery(channel, "SHOW TABLES")
  Tables_in_ripley
1        USArrests
> sqlQuery(channel, "DESCRIBE USArrests")
     Field        Type Null Key Default Extra
1     State varchar(255)  NO PRI     NA    NA
2   Murder       double YES         NA    NA
3  Assault      int(11) YES         NA    NA
4 UrbanPop      int(11) YES         NA    NA
5     Rape       double YES         NA    NA
```

`SHOW FULL TABLES` gives an additional additional column `Table_type`, the types of the tables/views.

There is useful information for end users in the `INFORMATION_SCHEMA` *database*, much more extensively as from MySQL 5.1.

Some of the non-standard behaviour can be turned off, e.g. starting MySQL with `--sql-mode=ANSI` gives closer conformance to the standard, and this can be set for a single session by

`SET SESSION sql_mode='ANSI'`

To change just the behaviour of quotes (to use double quotes in place of backticks) replace `ANSI` by `ANSI_QUOTE`.

The maximum size of a `char` column is 255 characters. That of a `varchar` column is up to 65535 characters (but there is a limit of 65535 bytes on the total size of a row), and those with a maximum of 255 or less are stored more efficiently. Types `text`, `mediumtext` and `longtext` can hold more, and are not subject to the row-size limit (`text` has default maximum size 65535, the default **RODBC** limit on transfers).

There are `binary`, `varbinary` and `blob` types which are very similar to their character counterparts but with lengths in bytes.

PostgreSQL

Table `pg_tables` lists all tables in all schemas; you probably want to filter on `tableowner='`*current_user*`'`, e.g.

```
> sqlQuery(channel, "select * from pg_tables where tableowner='ripley'")
  schemaname tablename tableowner tablespace hasindexes hasrules hastriggers
1     public     dtest     ripley         NA          0        0           0
```

There are both ANSI and Unicode versions of the ODBC driver on Windows: they provide many customizations. One of these is read-only access, another is if system tables are reported by `sqlTables`.

The default size of a `varchar` column is unlimited, but those with maximum length of 126 bytes or less are stored more efficiently. However, the ODBC interface has limits, which can be set in the configuration options. These include the maximum sizes for `varchar` (default 254) and `longvarchar` (default 8190), and how to handle unknown column sizes (default as the maximum), and whether 'Text' is taken as `varchar` or `longvarchar` (which affects the reported maximum size for a `varchar` column).

There is a single binary data type, `bytea`.

SQLite

These comments are only about SQLite 3.x.

Table `sqlite_master` lists tables and indices, and the `sql` column gives the SQL command used. E.g.

```
> tmp <- sqlQuery(channel, "select * from sqlite_master")
> tmp[, "sql"] <- substr(tmp[, "sql"], 1, 16)
> tmp
   type                       name  tbl_name rootpage              sql
1 table                   USArrests USArrests        2 CREATE TABLE "US
2 index sqlite_autoindex_USArrests_1 USArrests        4             <NA>
```

My current versions of Christian Werner's SQLite ODBC driver store character data in the current locale's charset (e.g. UTF-8) on Unix-alikes and by default in Unicode (UCS-2) on Windows (unless de-selected in the DSN configuration).

The default collation for text data is byte-by-byte comparisons, so avoid comparing non-ASCII character data in SQLite.

Actual Technologies sell an SQLite driver for OS X which requires `believeNRows = FALSE` and has a number of other issues including that it seems not to support dropping tables. (Christian Werner's SQLite ODBC driver was easy to install from the sources and worked correctly.)

Version of the SQLite ODBC driver since 0.87 have segfaulted on the test suite.

Oracle

Tables `cat`, `user_table` and `user_catalog` contain useful information on tables. Information on columns is in `all_tab_columns`, e.g.

```
> sqlQuery(channel,
          "select * from all_tab_columns where table_name='USArrests'")
  OWNER TABLE_NAME COLUMN_NAME DATA_TYPE DATA_TYPE_MOD
1 RIPLEY  USArrests       State  VARCHAR2            NA
2 RIPLEY  USArrests      Murder     FLOAT            NA
3 RIPLEY  USArrests     Assault    NUMBER            NA
4 RIPLEY  USArrests    UrbanPop    NUMBER            NA
5 RIPLEY  USArrests        Rape     FLOAT            NA
...
```

The Windows ODBC driver we tested had an option for a read-only connection.

Oracle's character data types are `CHAR`, `VARCHAR2` (character set specified when the database was created) and `NCHAR`, `NVARCHAR2` (Unicode), as well as `CLOB` and `NCLOB` for large character strings. For the non-Unicode types the units of length are either bytes or charactor (set as a default for the database) but can be overriden by adding a `BYTE` or `CHAR` qualifier. The limits are 4000 bytes apart from for `CLOB` and `NCLOB`, which have very high limits.

There are `RAW` and `BLOB` data types.

DB2

Schema `syscat` contains many views with information about tables: for example view `syscat.tables` lists all tables, and

```
> sqlQuery(channel,
          "select * from syscat.columns where tabname='USArrests'")
  TABSCHEMA    TABNAME  COLNAME COLNO TYPESCHEMA TYPENAME LENGTH SCALE
1  RIPLEY    USArrests    State     0    SYSIBM    VARCHAR    255     0
2  RIPLEY    USArrests   Murder     1    SYSIBM     DOUBLE      8     0
3  RIPLEY    USArrests  Assault     2    SYSIBM    INTEGER      4     0
4  RIPLEY    USArrests UrbanPop     3    SYSIBM    INTEGER      4     0
5  RIPLEY    USArrests     Rape     4    SYSIBM     DOUBLE      8     0
...
```

The `CHAR` type can have size up to 254 bytes: the maximum size of the `VARCHAR` type is 32762 bytes. For larger character strings there is the `CLOB` type (up to 2Gb). These types can be used to store data in a MBCS, including various Unicode encodings.

There are corresponding `BINARY`, `VARBINARY` and `BLOB` data types.

Microsoft SQL Server

There are several hundred views in schemas `INFORMATION_SCHEMA` and `sys` which will be listed by `sqlTables` and also by the stored procedure `sp_tables`. Another way to list tables is

```
SELECT * FROM sysobjects WHERE xtype='U'
```

where the condition restricts to user tables.

`USE` *database* changes the database in use.

Types `char` and `varchar` have a maximum specified size of 8000 bytes. It is possible to use `varchar(max)` (previously known as `text`) for a limit of 2Gb, but this may not work well with the ODBC interface. The Unicode types `nchar` and `nvarchar` have a maximum specified size of 4000 characters: again there is `nvarchar(max)` (formerly `ntext`).

There are corresponding `binary` and `varbinary` data types (with `image` as an earlier name for `varbinary(max)`).

Mimer

There are tens of views in schema `INFORMATION_SCHEMA` which can be read by SQL `SELECT` queries of the form

```
SELECT column-list
FROM INFORMATION_SCHEMA.view-name
WHERE condition
```

See the Mimer SQL Reference Manual chapter on Data Dictionary views for full details: two views are `TABLES` and `VIEWS`.

A session can be set to be read-only by the SQL command `SET   SESSION READ ONLY`.

Mimer uses Latin-1 for its default character types but Unicode types (`NCHAR` and `NVARCHAR`) are also available. Unsurprisingly given that the company is Swedish, different collations are allowed for both Latin-1 and Unicode character types.

The `char` and `varchar` columns have a maximum size of 15000 bytes: the `clob` data type is available for larger character columns. The `nchar` and `nvarchar` columns have a maximum size of 5000 characters: the `nclob` data type is available for larger Unicode columns.

There are corresponding `binary`, `varbinary` and `blob` binary data types.

A Installation

RODBC is simple to install, and binary distributions are available for Windows and OS X (before 10.9) from CRAN.

To install from the sources, an *ODBC Driver Manager* is required. Windows normally comes with one (it is part of MDAC and can be installed separately if required). OS X from 10.2 to 10.8 shipped with iODBC (http://www.iodbc.org, this is also available for other Unix-alikes) but from 10.9 the headers are no longer included in the OS X SDK. The CRAN OS X binary package (for 10.6 to 10.8) is compiled against iODBC.

For other systems the driver manager of choice is likely to be unixODBC, part of almost all Linux distributions and with sources downloadable from http://www.unixODBC.org. In Linux binary distributions it is likely that package unixODBC-devel or unixodbc-dev or some such will be needed.

Both unixODBC and iODBC can be installed from the sources under OS X: the latter requires configuring with --disable-gui. They need ODBC drivers compiled for the driver manager in use.

In most cases the package's configure script will find the driver manager files, and the package will install with no extra settings. However, if further information is required, use --with-odbc-include and --with-odbc-lib or environment variables ODBC_INCLUDE and ODBC_LIBS to set the include and library paths as needed. A specific ODBC driver manager can be specified by the --with-odbc-manager configure option, with likely values odbc or iodbc: if this is done for odbc and the program odbc_config is found, it is used to set the libpath as a last resort (it is often wrong), and to add any additional CFLAGS.

Sources of drivers

Keeping track of ODBC drivers is a never-ending task, and this section is no longer actively maintained. URIs are liable to move or disappear.

A list of drivers for unixODBC is maintained[15] at http://www.unixodbc.org/drivers.html. unixODBC ships with a number of drivers (although in most cases the DBMS vendor's driver is preferred)—these include for MySQL, PostgreSQL, Mimer and flat files.

MySQL provides drivers under the name 'Connector/ODBC' (formerly MyODBC') in source form, and binaries for all common 32-bit and most 64-bit R platforms. These are said to work also with MariaDB.

PostgreSQL has an associated project at http://pgfoundry.org/projects/psqlodbc/ and another project for at http://pgfoundry.org/projects/odbcng/. (Documentation for psqlodbc is currently hard to find, but there is some in the PostgreSQL 7.2 manual at http://www.postgresql.org/docs/7.2/static/odbc.html from before it was unbun-

---

[15]that the author works for Easysoft is conspicuous.

dled.) There are drivers for Unix-alikes and Windows – 64-bit Windows support is available as from PostgreSQL 9.0.

An SQLite ODBC driver for Unix-alikes, including OS X, and (32- and 64-bit) Windows is available from http://www.ch-werner.de/sqliteodbc/.

Oracle provides ODBC drivers as a supplement to its 'Instant Client' for some of its platforms (including 32/64-bit Windows and Linux but not currently OS X). See http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html One quirk of the Windows drivers is that the Oracle binaries must be in the path, so `PATH` should include e.g. `c:\Oracle\bin`.

For IBM's DB2, search its site for drivers for 'ODBC and CLI'. There are some notes about using this under Linux at http://www.unixodbc.org/doc/db2.html.

Mimer (www.mimer.com) is a cross-platform DBMS with integral ODBC support, so

> 'The Mimer SQL setup process automatically installs an ODBC driver when the Mimer SQL client is installed on any Windows or UNIX platform.'

The 'HowTos' at http://developer.mimer.se/howto/index.tml provide some useful hints.

Some details of the 32-bit Microsoft 'ODBC Desktop Database Drivers' (for Access, Excel, Paradox, dBase and text files on Windows) can be found at http://msdn.microsoft.com/en-us/library/ms709326%28VS.85%29.aspx. There is also a Visual FoxPro driver and an (outdated) Oracle driver.

32-bit Windows drivers for Access 2007 and Excel 2007 are bundled with Office 2007 but can be installed separately *via* the installer `AccessDatabaseEngine.exe` available from http://www.microsoft.com/en-us/download/details.aspx?id=23734.

The Access/Excel 2010 versions at http://www.microsoft.com/en-us/download/details.aspx?id=13255 have a 64-bit version: however the 64-bit drivers cannot be installed alongside 32-bit versions of Office (as far as we know, and definitely not for Office 2007).

For recent versions of OS X, low-cost and easy-to-use `iODBC` drivers are available from http://www.actualtech.com/products.php: these cover MySQL/PostgreSQL/SQLite (one driver), SQL Server/Sybase, Oracle, and a read-only driver for Access and related formats (including Access 2007 and Excel, but not Excel 2007). That SQLite driver needs `believeNRows = FALSE` set. Currently at least, installing those drivers on 10.9 installs `iODBC`.

OS X drivers for the MySQL, PostgreSQL and the major commercial databases are available from http://uda.openlinksw.com/.

Specifying ODBC drivers

The next step is to specify the ODBC drivers to be used for specific DBMSs. On Windows installing the drivers will register them automatically. This might happen as part of the

```
$ cat /etc/odbcinst.ini
[MySQL]
Description    = ODBC 3.51.26 for MySQL
Driver         = /usr/lib64/libmyodbc3.so
FileUsage      = 1

[MySQL ODBC 5.1 Driver]
Description     = ODBC 5.1.05 for MySQL
Driver          = /usr/lib64/libmyodbc5.so
UsageCount      = 1

[PostgreSQL]
Description     = ODBC for PostgreSQL
Driver          = /usr/lib64/psqlodbc.so
FileUsage       = 1

[sqlite3]
Description = sqliteodbc
Driver = /usr/local/lib64/libsqlite3odbc.so
Setup = /usr/local/lib64/libsqlite3odbc.so
FileUsage = 1
```

Figure 1: A system ODBC driver file from a `x86_64` Fedora 10 Linux system using `unixODBC`.

installation on other systems, but usually does not.

Both `unixODBC` and `iODBC` store information on drivers in configuration files, normally system-wide in `/etc/odbcinst.ini` and per-user in `~/.odbcinst.ini`. However, the system location can vary, and on systems with `unixODBC` can be found by at the Unix command line by one of

```
$  odbcinst -j
$  odbc_config --odbcinstini
```

For `iODBC` use `iodbc_config`: on OS X the system location used by Apple was `/Library/ODBC/odbcinst.ini`.

The format can be seen from figure 1. (`unixODBC` allows `Driver64` here to allow for different paths on 32-bit and 64-bit platforms sharing a file system.) The MySQL and PostgreSQL drivers were installed from the Fedora RPMs `mysql-connector-odbc` and `postgresql-odbc`, and also from the `mysql-connector-odbc` RPM in the MySQL distribution (which inserted the entry in the driver file).

The MySQL manual gives detailed information (including screenshots) of installing its drivers and setting up DSNs that may also be informative to users of other DBMSs.

B   Specifying DSNs

The ODBC driver managers have 'User DSNs' and 'System DSNs': these differ only in where the information is stored, the first on a per-user basis and the second for all users of the system.

Windows has a GUI[16] to set up DSNs, called something like 'Data Sources (ODBC)' under 'Administrative Tools' in the Control Panel. You can add, remove and edit ('configure') DSNs there (see figure 2). When adding a DSN, first select the ODBC driver and then complete the driver-specific dialog box. There will usually be an option to test the DSN and it is wise to do so.

If `Rgui` is to be used on Windows, incomplete DSNs can be created and the dialog box will be brought up for completion when `odbcConnect` is called—this can be helpful to avoid storing passwords in the Windows Registry or to allow alternate users or databases. On that platform, calling `odbcDriverConnect()` with no arguments will bring up the main ODBC Data Sources dialog box to allow a DSN to be constructed on the fly.

OS X prior to 10.6 came with a very similar GUI (figure 3) found at Applications / Utilities / ODBC Administrator. This has been available as a download from http://support.apple.com/kb/DL895. Another GUI for OS X is available from http://www.odbcmanager.net/index.php.

On Unix-alikes DSNs can also be specified in files (and the graphical tools just manipulate these files). The system-wide file is usually `/etc/odbc.ini` and the per-user file[17] `~/.odbc.ini`. Some examples of the format are shown figure 5.

What fields are supported is driver-specific (and it can be hard to find documentation). There is no clear distinction between fields that specify the driver and those which specify the DSN, so any parts of the driver specification which might differ between connections can be used in the DSN file.

Things that are often set here are if the connection is read-only (`test_pg` is *not* readonly) and the character encoding to be used.

Command-line programs `isql` (`unixODBC`) and `iodbctest` (`iODBC`) can be used to test a DSN that has been created manually in a file. The formats are

```
$ isql -v dsn db˙username db˙password
$ iodbctest
```

Both give a command-line SQL interface: use `quit` to terminate.

---

[16]Extra care is needed on a 64-bit version of Windows, as this GUI shows only 64-bit settings for ODBC, including drivers and DSNs. If you are running 32-bit R (and hence 32-bit ODBC) on 64-bit Windows, you need the 32-bit version of the GUI at something like `c:\Windows\SysWOW64\odbcad32.exe` – and beware that both 32- and 64-bit versions are called `odbcad32.exe`.

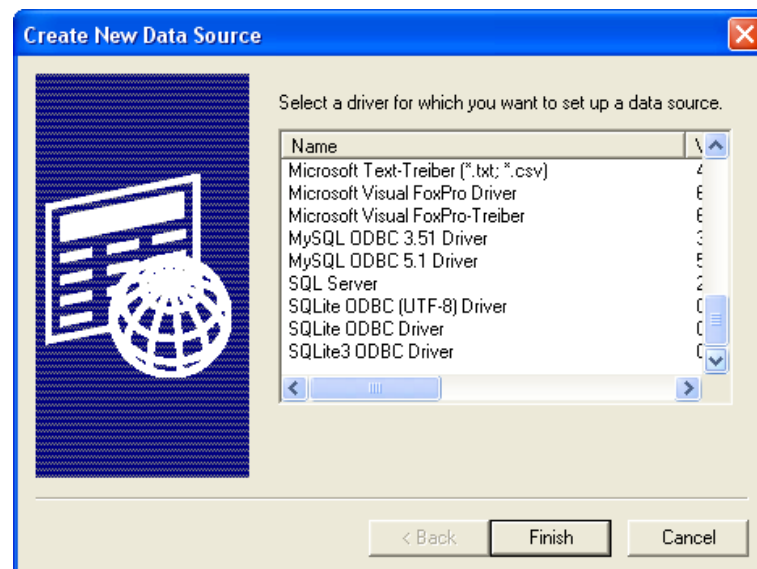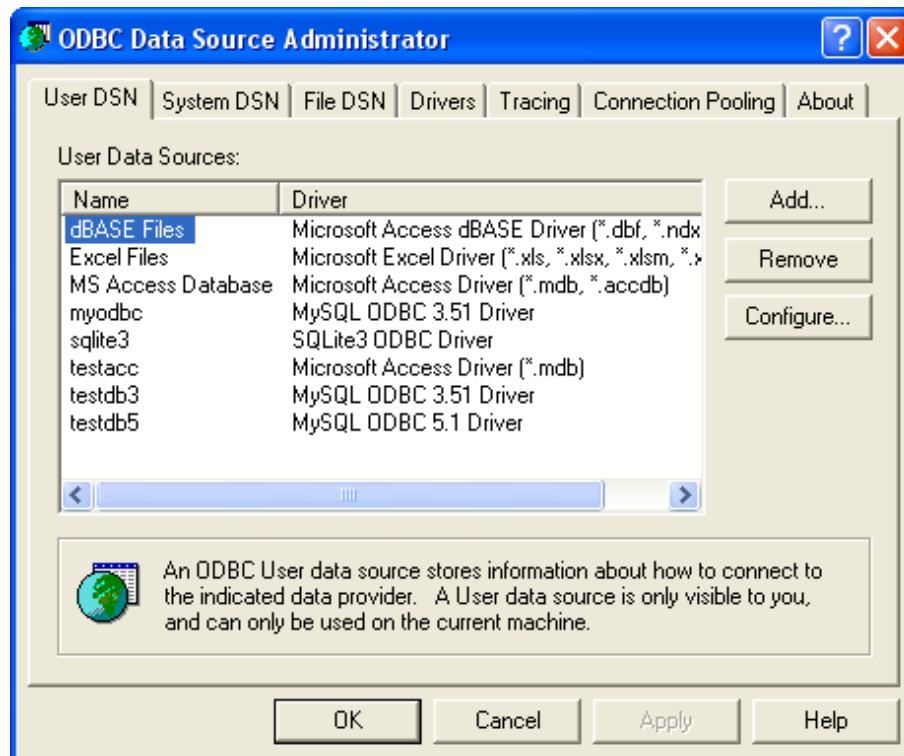[17]`~/Library/ODBC/odbc.ini` on Mac OS X.

Figure 2: (Top) The main Data Sources (ODBC) dialog box from a Windows XP system. (Bottom) The dialog box to select a driver that comes up when the Add button is clicked.
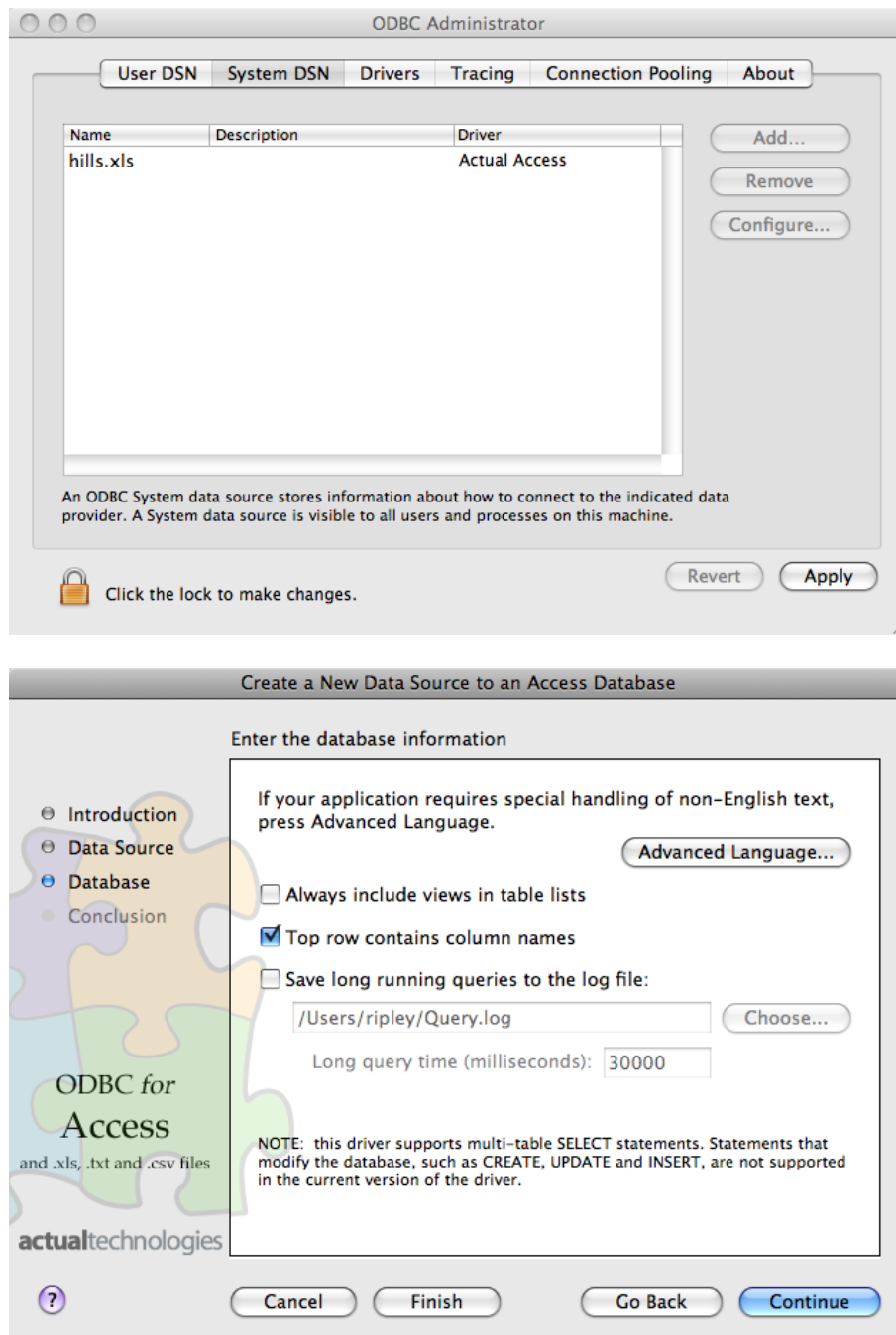
Figure 3: (Top) The main ODBC Administrator dialog box from a OS X 10.5 system. (Bottom) A page of the dialog box to specify a DSN for the Actual Technologies Access/Excel driver.
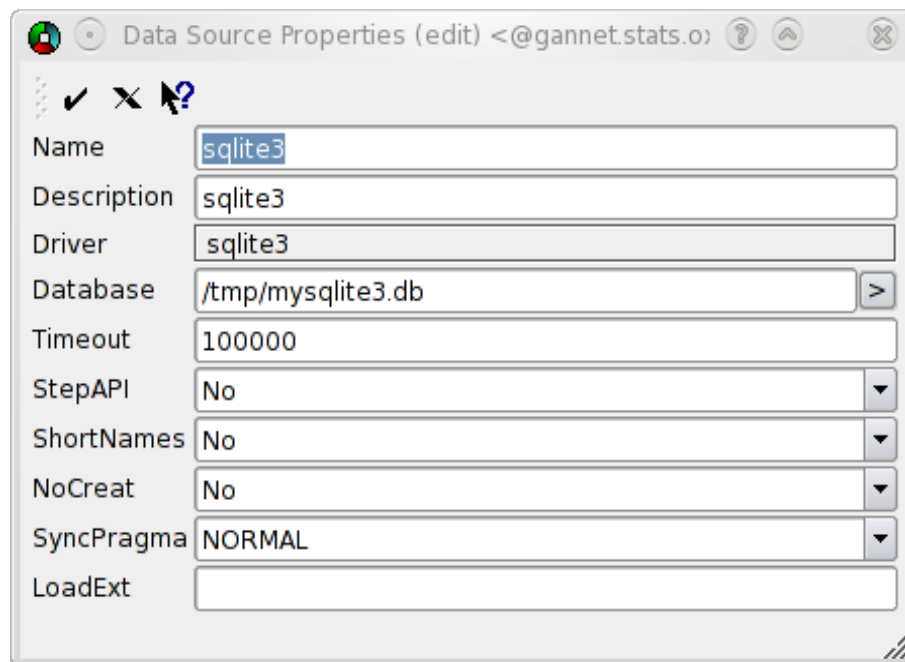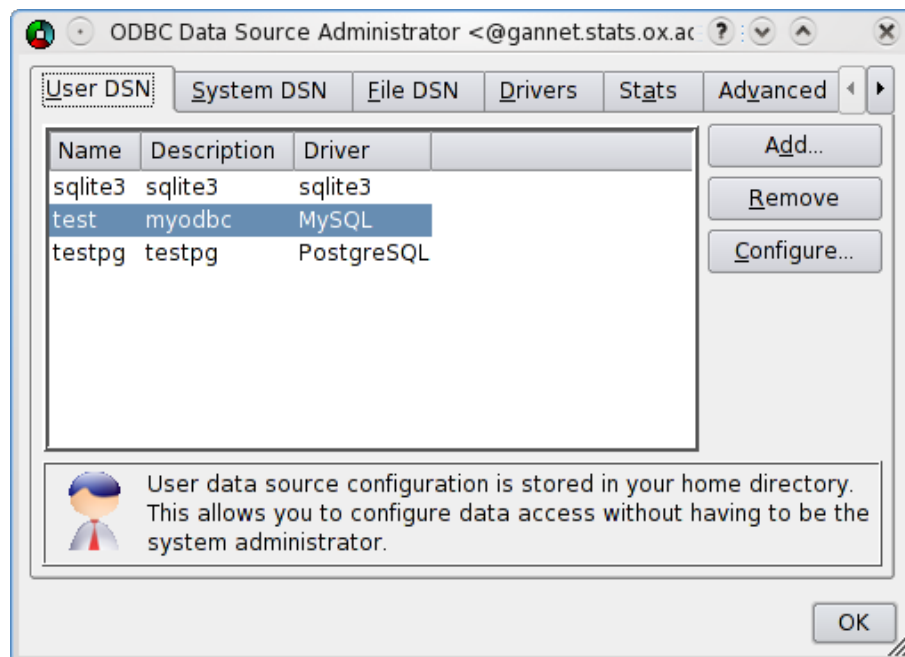
Figure 4: The dialog box of `ODBCconfig` on Fedora 10 Linux, and the `Configure` screen for the SQLite driver.

```
[test_mysql]
Description     = test MySQL
Driver          = MySQL
Trace           = No
Server          = localhost
Port            = 3306
Database        = test

[test_mysql5]
Description     = myodbc5
Driver          = MySQL ODBC 5.1 Driver
Server          = gannet
Port            = 3306
Database        = ripley

[test_pg]
Description     = test PostgreSQL
Driver          = PostgreSQL
Trace           = No
TraceFile       =
ServerName      = localhost
UserName        = ripley
Port            = 5432
Socket          =
Database        = testdb
ReadOnly        = 0

[test_sqlite3]
Description     = test SQLite3
Driver          = sqlite3
Database        = /tmp/mysqlite3.db
```

Figure 5: A personal (`~/.odbc.ini`) file from a Fedora 10 Linux system using `unixODBC`.
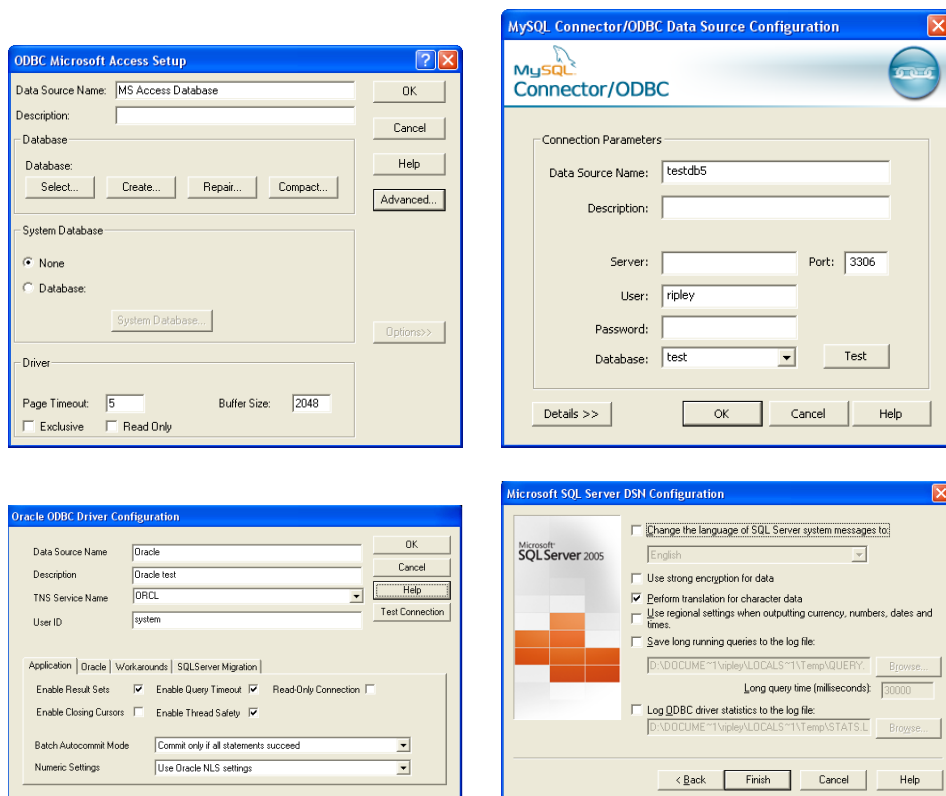
Figure 6: Parts of the ODBC driver configuration screens on Windows XP for Microsoft Access, MySQL Connector/ODBC 5.1, Oracle's ODBC driver and Microsoft SQL Server.

C Internals

The appendix is in part an *aide memoire* for the maintainer, but may interest the curious user.

RODBC connection objects are an integer with several attributes: they are numbered consecutively in the current session. For example

```
> channel <- odbcConnect("test")
> unclass(channel)
[1] 1
attr(,"connection.string")
[1] "DATABASE=ripley;DESCRIPTION=myodbc;DSN=test;OPTION=0;PORT=3306;SERVER=localhost;"
attr(,"handle_ptr")
<pointer: 0x233e6c0>
attr(,"case")
[1] "nochange"
attr(,"id")
[1] 11371
attr(,"believeNRows")
[1] TRUE
attr(,"colQuote")
[1] "`"
attr(,"tabQuote")
[1] "`"
attr(,"encoding")
[1] ""
attr(,"rows_at_time")
[1] 100
attr(,"isMySQL")
[1] FALSE
```

Most of the attributes record the arguments of `odbcDriverConnect`. The `"connection.string"` attribute is as returned by `SQLDriverConnect` and list driver-specific parameters separated (and perhaps terminated) by a semicolon. The `"id"` attribute is a random integer used for integrity checks (and in particular to reject connection objects should they be saved and restored in a different session). The `"isMySQL"` attribute is used both to select the default quote character and the interpretation of `qualifier.table` names.

The main structure of the connection is kept as a C `struct`, a pointer to which is passed around as the R external pointer `"handle_ptr"`. This has a finalizer that will close the connection when there is no longer an R object referring to it (including at the end of the R session), with a warning unless the connection has already been closed by `close` or `odbcClose`. In addition, a C-level table keeps the pointers of the first 1000 connections of an R session, to enable `odbcCloseAll` to close them.

The `struct` is currently defined as

```
typedef struct rodbcHandle {
    SQLHDBC     hDbc;       /* connection handle */
    SQLHSTMT    hStmt;      /* statement handle */
    SQLLEN      nRows;      /* number of rows and columns in result set */
    SQLSMALLINT nColumns;
    int         channel;    /* as stored on the R-level object */
    int         id;         /* ditto */
    int         useNRows;   /* value of believeNRows */

    /* entries used to bind data for result sets and updates */
```

```
    COLUMNS     *ColData;
    int         nAllocated;
    SQLUINTEGER rowsFetched;  /* use to indicate the number of rows fetched */
    SQLUINTEGER rowArraySize; /* use to indicate the number of rows we expect back */
    SQLUINTEGER rowsUsed;     /* for when we fetch more than we need */

    SQLMSG      *msglist;     /* root of linked list of messages */
    SEXP        extPtr;       /* the external pointer address */
} RODBCHandle, *pRODBCHandle;
```

Most ODBC operations work by sending a query, explicitly or implicitly via e.g.
`sqlColumns`, and this creates a *result set* which is transferred to an R data frame by
`sqlGetResults`. `nRows` and `nCols` indicate the size of the pending result set, with
`nCols = -1` used if there are no pending results.

ODBC works with various *handles*. There is a `SQLHENV` handle for the environment that
RODBC opens when a connection is first opened or DSNs are listed—its main use is to
request ODBC 3 semantics. Then each connection has a `SQLHDBC` handle, and each query
(statement) a `SQLHSTMT` handle. Argument `literal=TRUE` of `sqlTables` and `sqlColumns`
is used to set the `SQL_ATTR_METADATA_ID` attribute of the statement handle to be true.

All the functions[18] that create a result set call C function `cachenbind`. This allocates
buffers under the `colData` pointer and binds the result set to them by `SQLBindCol`. Then
when `sqlGetResults` calls the C function `SQLFetch` or `SQLFetchScroll` the results for
one or more (up to `MAX_ROWS_FETCH = 1024`) rows are loaded into the buffers and then
copied into R vectors.

Prior to **RODBC** 1.3-0 the default was to fetch a row at a time, but it is now to fetch up
to 100 rows at a time. Entries `rowsArraySize` and `rowsFetched` are used to indicate how
many rows were requested and how many were available. Since e.g. `sqlFetch` allows a
maximum number of rows to be returned in the data frame, `rowsUsed` indicates how many
of the rows last fetched have so far been returned to R.

The buffers are part of the `ColData` entry, which is an array of `COLUMNS` structures, one of
each column in the result set. These have the form

```
typedef struct cols {
    SQLCHAR     ColName[256];
    SQLSMALLINT NameLength;
    SQLSMALLINT DataType;
    SQLULEN     ColSize;
    SQLSMALLINT DecimalDigits;
    SQLSMALLINT Nullable;
    char        *pData;
    int         datalen;
    SQLDOUBLE   RData [MAX_ROWS_FETCH];
    SQLREAL     R4Data[MAX_ROWS_FETCH];
    SQLINTEGER  IData [MAX_ROWS_FETCH];
    SQLSMALLINT I2Data[MAX_ROWS_FETCH];
    SQLLEN      IndPtr[MAX_ROWS_FETCH];
} COLUMNS;
```

The first six entries are returned by a call to `SQLDescribeCol`: `DataType` is used to select
the buffer to use. There are separate buffers for double-precision, single-precision, 32-bit
and 16-bit integer and character/byte data. When character/data buffers are allocated,

---

[18] `odbcQuery`, `sqlColumns`, `sqlPrimaryKeys`, `sqlTables` and `sqlTypeInfo`.

`datalen` records the length allocated per row (which is based on the value returned as `ColSize`). The `IndPtr` value is used to record the actual size of the item in the current row for variable length character and binary types, and for all nullable types the special value `SQL_NULL_DATA` (-1) indicates an SQL null value.

The other main C-level operation is to send data to the ODBC driver for `sqlSave` and `sqlUpdate`. These use `INSERT INTO` and `UPDATE` queries respectively, and for `fast = TRUE` use parametrized queries. So we have the queries (split across lines for display)

```
> sqlSave(channel, USArrests, rownames = "State", addPK = TRUE, verbose = TRUE)
Query: CREATE TABLE "USArrests"
  ("State" varchar(255) NOT NULL PRIMARY KEY, "Murder" double, "Assault" integer,
  "UrbanPop" integer, "Rape" double)
Query: INSERT INTO "USArrests"
  ( "State", "Murder", "Assault", "UrbanPop", "Rape" ) VALUES ( ?,?,?,?,? )
Binding: 'State' DataType 12, ColSize 255
Binding: 'Murder' DataType 8, ColSize 15
Binding: 'Assault' DataType 4, ColSize 10
Binding: 'UrbanPop' DataType 4, ColSize 10
Binding: 'Rape' DataType 8, ColSize 15
Parameters:
...

> sqlUpdate(channel, foo, "USArrests", verbose=TRUE)
Query: UPDATE "USArrests" SET "Assault"=? WHERE "State"=?
Binding: 'Assault' DataType 4, ColSize 10
Binding: 'State' DataType 12, ColSize 255
Parameters:
...
```

At C level, this works by calling `SQLPrepare` to record the insert/update query on the statement handle, then calling `SQLBindParameter` to bind a buffer for each column with values to be sent, and finally in a loop over rows copying the data into the buffer and calling `SQLExecute` on the statement handle.

The same buffer structure is used as when retrieving result sets. The difference is that the arguments which were ouptuts from `SQLBindCol` and inputs to `SQLBindParameter`, so we need to use `sqlColumns` to retrieve the column characteristics of the table and pass these down to the C interface.