DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

R/S Interfaces to Databases

Torsten Hothorn,* Friedrich-Alexander-Universität Erlangen,
Torsten.Hothorn@rzmail.uni-erlangen.de
David A. James, Bell Labs, Lucent Technologies, dj@bell-labs.com
Brian D. Ripley, University of Oxford, ripley@stats.ox.ac.uk

Abstract

Connectivity is an increasingly important part of statistical computing, and interfaces to databases are becoming important both in large-scale data mining applications and from the use of smaller personal databases.

In this paper we present current work and future plans on interfacing the S language (R and S-PLUS) to databases, in particular to relational database management systems (DBMS).

1 Introduction

In this paper we present current work and future plans on interfacing the S language (R and S-PLUS) to databases, particularly to relational database management systems (DBMS). We discuss the interface to databases from the following perspectives:

- DBMS in the context of distributed computing, not simply as data repositories, but bona fide systems with a language plus concurrency, persistence, events and security models — in addition to distributed and parallel computations.
- Interfaces to databases as yet-another application of inter-system communications. As such, we will be able to take advantage of this client-server technology, but we will also have to address the problems that it raises, such as finalization of remote objects, concurrency, function call-backs to S, event handling, asynchronous communications, the possible need for either multiple evaluators and/or S threads, etc.

*Support by Deutsche Forschungsgemeinschaft SFB 539 is gratefully acknowledged.

These issues surface not only in the context of database interfaces, but also in the development of graphical user interfaces [4], communication with Microsoft Excel [10], and in general inter-system communications [8].

- We approach interfaces to databases in the context of data analysis, and not so much as the development of online processing or database development. We emphasize the connectivity to databases as a necessity to carry unfettered data analysis [12]. We emphasize the need to create a common interface so that further more advanced tools may be built on top of this common API.

In Section 2, Brian Ripley provides a bird-eye's view of current solutions in R and S-Plus, their implementations, and their shortcomings. Some communication has been published regarding this topic: the R Data Import/Export manual [14] discussed recent implementations and article [13] focuses on usage issues, illustrated by the `RODBC` package.

In Section 3, Torsten Hothorn presents a vision of the future from a user's point of view — what is needed and possible how it would benefit users. In Section 4 David James presents a vision of the future from a provider's perspective.

2 Current Solutions

'Databases' are more properly known as DBMSs (DataBase Management Systems). There are several types including

- Text and `csv` files.
- Spreadsheets (programs and formats), like *Excel*.
- Flat file databases like *DBase*.
- Hierarchical databases (such as HDF5, `http://hdf.ncsa.uiuc.edu`).
- 'Lean' relational databases like *Access* and *MySQL*, *MiniSQL* (also known as *mSQL*).
- Heavyweight relational databases like *Oracle*, *DB/2*, *Informix*, *Sybase*, *SQL Server*, *PostgreSQL*.

Sometimes the user has a choice of DBMS, but often has to use an existing database and hence DBMS. Most of the relational DBMSs are client–server systems, and many allow communication over TCP/IP.

Most DBMSs come with a *monitor*, a text-based client, and some have GUI clients (notably Access, which looks very like a set of spreadsheets). Almost all have a C or C++ API. And they are almost all different.

The actual commands are normally sent in a dialect of SQL (Structured Query Language). This has ISO standards, rarely complied with. (Having multiple standards with multiple levels encourages non-compliance!)

R interfaces

There are three DBMS-specific interface packages on CRAN,

`RPgSQL`	for *PostgreSQL*	by Tim Keitt
`RMySQL`	for *MySQL*	by David James & Saikat DebRoy
`RmSQL`	for *MiniSQL*	by Torsten Hothorn

using the C interfaces. All three run successfully on Linux, and we have run `RMySQL` on Windows, with some difficulty.

These and `RODBC` are described in the *R Data Import/Export* manual [14].

S-PLUS interfaces

S-PLUS 5.1/6.0 on Unix (and not Linux) have connections to import data from *Informix*, *Oracle* and *Sybase* databases.

`RMySQL` has a sibling `SMySQL` and a cousin `SOracle`[1] connecting to Oracle databases.

Cross-DBMS solutions

There are a number of cross-database interfaces. ODBC (Open DataBase Connectivity) is an X/Open and ISO standard for a common interface to relational databases. It is common in the Windows world, and has been much enhanced by Microsoft [9] and used as the basis of later developments (ADO, ...). On Windows, spreadsheets and even text files are covered. One of the Unix ODBC driver managers (`unixODBC`, `http://www.unixodbc.org`) has a text-file driver. ODBC drivers are available for all the common DBMSs (not all freely, and more widely on Windows than on Linux). CRAN has a package `RODBC` (by Michael Lapsley) which interfaces to ODBC driver managers on Windows and Linux/Unix.

Two other cross-database solutions worth noting are JDBC[2] (Java), [7, 11]) and Perl DBI[3] [5]. Either could be used via the Omegahat R/S interfaces to those languages, and Saikat DebRoy has been working on an RS-JDBC package.

2.1 Retrieving Data

Perhaps this is the most common task of all. A *table* in a relational DBMS is a similar concept to a data frame in S, so a natural idea is to (in principle, perhaps) create a table containing the required data, and map it to a data frame.

`RmSQL` does this row by row. The others have interfaces like

```
> db.connect(dbname="testdb")                    # RPgSQL
> df <- db.read.table("USArrests")

> channel <- odbcConnect("testdb"")              # RODBC
> sqlFetch(channel, "USArrests", rownames = TRUE)
```

[1] both available at `http://www.omegahat.org/download/contrib/RS-DBI`.
[2] `http://java.sun.com/products/jdbc/`.
[3] `http://dbi.symbolstone.org/`

```
> con <- dbConnect(MySQL(), dbname = "test")    # RMySQL
> getTable(con, "arrests")        # leaves row names as a column
```

hiding the SQL which they generate from the user. Unfortunately, snags arise quite soon.

1. **Case**. Few DBMSs are case-sensitive. S is. This applies not just to table names, but also to column names. Which case is used is DBMS- and even OS-specific (e.g. *MySQL* is case-sensitive on Linux and case-insensitive on Windows).

2. **Row-ordering**. Relational DBMSs have in principle unordered tables. You do need to retrieve an ID field (primary key) to act as the row names.

3. The mapping of **data types** to R data types is problematic (see later).

4. Representation of **missing values**. These are commonly represented as the SQL value `NULL`, but watch out for the differences between that and `""`, and how they get transferred.

5. **Size**. Retrieving the whole table at once might be too much for R to hold. You might need to retrieve a block of rows or columns at the time, and for that you will probably need to use SQL.

2.2 Uploading Data

For some purposes, uploading data to a database is an important task. Convenience functions are available in most of the R interfaces:

```
> data(USArrests)
> usarrests <- USArrests
> db.connect(dbname="testdb")                    # RPgSQL
> db.write.table(usarrests, write.row.names = TRUE)

> channel <- odbcConnect("testdb"")              # RODBC
> sqlSave(channel, USArrests, rownames="state")

> con <- dbConnect(MySQL(), dbname = "test")    # RMySQL
> assignTable(con, "arrests", USArrests, overwrite=TRUE)
```

We have to ensure that the row names get saved, if (as here) they are the ID field.

The alternative is to write a file and load that via SQL, either from the interface or the monitor program. That is complex and proves to be error-prone.

Data Types

Mapping S data types to the DBMS's data types is tricky, and only `RPgSQL` seems to make a good job of it. One issue is what to do if a table of the same name exists. Currently `RODBC` updates a table if it looks suitable, otherwise destroys it and writes a table entirely as character fields.

Mapping in the other direction (from the DBMS to S) is perhaps easier, as data frames are created anew. The details of the existing interfaces are largely undocumented: `RODBC` for example reads the result set as a character matrix and converts it, as a `read.table`–alike.

One big issue for a cross-DBMS solution is finding out the data types that the DBMS supports, what they are called and what C/R type they correspond to. ODBC has commands to do this, but unfortunately some drivers[4] appear cavalier about this.

`RMySQL` has recently added a function `SQLDataType` to find the DBMS data type that can most closely represent an R/S object.

2.3 Sending SQL Commands

All of the interfaces allow SQL commands to be sent to the DBMS, for example

```
> db.connect(dbname="testdb")                    # RPgSQL
> db.execute("SELECT rpgsql_row_names, murder FROM usarrests",
             "WHERE rape > 30 ORDER BY murder", clear=FALSE)
> db.fetch.result()

> channel <- odbcConnect("testdb"")              # RODBC
> sqlQuery(channel, "select state, murder from USArrests
           where rape > 30 order by murder")

> con <- dbConnect(MySQL(), dbname = "test")     # RMySQL
> quickSQL(con,
   "select * from arrests where Rape > 30 order by Murder")
```

There are many dialects of SQL (official and unofficial) and the names of the data types differ widely too. ODBC makes some effort to smooth over the differences.

We have already seen that `RPgSQL` and `RODBC` go some way to encapsulating common operations in intuitive functions, as do the latest versions of `RMySQL`. `RPgSQL` goes further. It has convenience functions `sql.select` and `sql.insert` for the simpler cases of common operations. These are particularly useful on a system with transaction support (like *PostgreSQL*). We would like to see a lot more high-level functionality, hiding issues like case and data types from the users, transparently capturing row names and marking them as a primary key, mapping column names to and from those supported by the DBMS ....

One tremendous simplification, at least superficially, is *proxy objects*. Tim Keitt has implemented the very appealing notion of *proxy* data frames in `RPgSQL`. These are R objects of a class `"db.proxy"` that in most respects behaves like `"data.frame"`, but are really references to tables in the database. For example

```
> db.connect(dbname="testdb")                    # RPgSQL
> bind.db.proxy("USArrests")
## USArrests is now a proxy, so all accesses are to the database
> USArrests[, "Rape"]
```

[4] notably `MyODBC`

```
   Rape
1  21.2
2  44.5
   ...
> rm(USArrests) # remove proxy
```

Unfortunately, row names are not handled as transparently as by `db.read.table`.

The class has methods for operators `[, $, [[` and functions `dim`, `names`, `row.names` and `dimnames`, as well as `print`, `summary` and coercion functions to matrix, list and data frame.

There are limits, though, so for example `USArrests[rows, "Rape"]` will only work if `rows` specifies an interval of numeric row indices (by any S indexing scheme). At present these proxy data frames are read-only, so there are no replacement functions such as `[<-`. Presumably one would need to implement batching to make such operations efficient in a transactional DBMS.

2.4 Functions in Databases

All the DBMSs allow some possibility for using functions and operators in SQL statements. There are lots of differences: for example *PostgreSQL* and *MySQL* convert data types suitably, but *Access* does not. The names of functions, their exact scope and whether user-defined functions are allowed all differ widely.

Access uses Visual Basic for Applications; *MySQL* and *PostgreSQL* allow user-defined functions. For all three this opens the possibility of embedding R functionality in user-defined functions, via DCOM for *Access* on Windows and via a shared library on Linux/Unix for *MySQL* and *PostgreSQL*. (Duncan Temple Lang has explored the latter approach, and code is available at `http://www.omegahat.org`.)

2.5 Current Status

For R users, the following summarizes the position in March 2001.

- Macintosh users have no choice!
- Windows users have the 'Ford Model T' choice. There is only one easy solution: use `RODBC`. (`RMySQL` and *MySQL* can be used with some work. In principle `RPgSQL` could be used with *PostgreSQL*, but we have not succeeded in doing so.)
- If your data are on a commercial DBMS, you have (at most) one choice, `RODBC`.
- If you can choose your DBMS, and you are on Unix/Linux, you may consider `RPgSQL` and *PostgreSQL* or `RMySQL` and *MySQL*.

3 Future Directions: The Users' Perspective

In all fields of statistics modern data management is nowadays based on database management systems, mostly relational ones. They help us keeping data organized, secure and

accessible, even in multi-user environments. Therefore, the statistician (and R user) is confronted with these systems if (s)he needs to extract or insert data. We therefore want simple methods to access data without the need of experienced programming. The main target is independence from the database engine itself. Other languages provide well-defined programming interfaces to relational DBMSs, for example JDBC, ODBC, Perl DBI and others. In contrast, R is not only a programming language but a language for data analysis. So one does not only need a programming interface but a user interface to DBMSs. We have to find out how an interface should look like considering what applications or problems R users are confronted with.

In the following, we introduce a simple example of application from medical statistics and discuss the needs of users confronted with similar problems. Furthermore, we outline applications as data mining and discuss possibilities for the design of a common and simple but powerful R-DBMS user-interface.

3.1 Example: The Erlangen Glaucoma Database

Glaucoma is a irreversible neurodegenerative eye disease and one of the main reasons for blindness. Starting in 1991, several projects at the Erlangen Eye Department formed a collection of cases and normals, resulting in the Erlangen Glaucoma Database. For sake of simplicity, we translate the German database definition to English. Each eye of a patient or control subject is identified by a patient number (`patnr`), the examination number (`examnr`) and the eye-side (`eyeside`).

If a statistician wants to work with this database, the first problems occur far before any system for statistical analysis is involved. Usually, the database was defined by other people a long time ago and is maintained on systems not administrated by the statistician himself. Therefore, details are needed about the database structure and it is crucial to know at least a username and password as well as the host which the DBMS engine is running on.

As a simple example, consider the following situation. One wants to compute the median of the intraocular pressure (`iop`) of all patients or control subjects. Currently with `RMySQL`, this can be done using following code:

```
library(RMySQL)
m <- MySQL()
con <- dbConnect(m, host = "artemis", user = "hothorn",
                 password="secret", dbname = "glaucoma")
iop <- quickSQL(con, stmt)
median(iop$iop, na.rm=T)
```

This simple example shows the complexity of one of the current interfaces. First, the user needs to specify access information: the host it runs on, a username and password and the name of the database. To form a correct SQL statement, at least the name of the relation (`examination`) and the variable name (`iop`) is needed. Most important: without a basic knowledge of SQL there is no chance of success.

One may think of a relation as of a (possibly huge and fast) data frame. But how serious is this? Usually, databases consist of many relations linked together using keys (which in our example are `patnr`, `examnr` and `eyeside`). So a more realistic example would be:

draw two histograms of the intraocular pressure, one for the normal subjects, one for the glaucoma subjects. To solve this problem, a more detailed knowledge of SQL is needed.

```
stmt <- "SELECT examination.iop, diagnosis.diagn FROM
    examination, diagnosis
    WHERE examination.patnr = diagnosis.patnr AND
    examination.examnr = diagnosis.examnr AND
    examination.eyeside = diagnosis.eyeside AND
    diagnosis.diagn = \"normal\" "
iopnormal <- quickSQL(con, stmt))
```

Here, the information about the diagnosis of a subject is stored in relation `diagnosis`, field `diagn` and both `iop` and `diagn` are returned after joining the two relations `diagnosis` and `examination`.

Even more questions arise. Suppose that the statistician is allowed to update misspecified values. Clearly, this could be done by formulating an update statement in SQL, but what about different users interacting on the same data set? Once that some users simultaneously write data to the system, transactional support is needed.

3.2 Three Types of Users

This subsection tries to 'classify' users by their needs and knowledge about database systems and, more important, user interfaces. In our opinion, one can identify at least three types of users with completely different needs and views to database systems.

3.2.1 The Sophisticated Users

The sophisticated users are the statisticians with detailed knowledge of relational databases. Often they use or create relational databases with the engine they are comfortable with. They are familiar with SQL and tend to be quite sophisticated to know how to avoid obvious problems, such as trying to import a huge table all at once or type conversions, so they can take care of themselves. Most of the times, they want direct access to the database engine itself. The specific interfaces currently available, for example `RMySQL` or `RPgSQL`, fit their needs. However, changing the database engine causes trouble, so a unified interface is needed.

3.2.2 The Unsophisticated Users

SQL is not straightforward to use and most people do not want to create SQL statements. There is a need of structures like data frames. The following code is varporware:

```
con <- dbConnect("MySQL", group = "glaucoma")
proxyGlaucoma <- proxyRDBMS(con)
df <- as.data.frame(examination)
median(df$iop)
```

Clearly, a user familiar with data.frames has no problems dealing with `df`. A very similar approach is the use of `bind.db.proxy` in `RPgSQL`. From this point of view, a relation in a

database is a huge and fast data frame. The advantage of this approach is that only the data needed for the computations is transfered. On the other hand, real databases always are built up from relations linked together using keys and that means that they are not just big data frames. Powerful queries consist of joins. How can one offer this functionality using proxy objects? Another problem is that on the one hand, users are often not allowed to create their own relations in a client-server environment. On the other hand, an inexperienced user may[5] have problems setting up his own database engine. Therefore, a simple way of using a database as an 'extended memory' through proxy objects may have some drawbacks.

We also have to focus on an application of an R-DBMS user-interface completely different from the one discussed until now: data mining.

3.2.3 The Data Miner

Having really big datasets at hand may bring even R into trouble. One can think of using the DBMS as memory, for example for fitting linear models to large datasets. This issue is discussed e.g. in [12] and [8].

3.3 Programming Interfaces

Well-established interfaces as JDBC, ODBC or Perl DBI are programming interfaces, but not user interfaces. However, they are similar to the interfaces available for R in the following sense. They offer system-independent functionality for

- connecting to a database engine,
- user authentication,
- executing an SQL query,
- fetching the results (including type conversions),
- error handling.

As an example, the query selecting the intraocular pressure for all subjects would be formulated in Perl DBI as:

```
use DBI;
$dbh = DBI->connect("DBI:mysql:database=glaucoma",
                    "hothorn", "secret",
                    { RaiseError => 1, AutoCommit => 0 });
$sth = $dbh->prepare("SELECT iop FROM examination");
$sth->execute();
while ( @row = $sth->fetchrow_array ) { print "@row\n"; }
$dbh->disconnect;
```

This is very close to the current status of RMySQL. As outlined, the sophisticated user is comfortable with a solution just providing access to SQL.

[5] although such users do seem to find `Access` accessible.

3.4 User Interfaces

The crucial question is: 'Do we need SQL for the user interface?'. Some ideas for hiding SQL are available, for example proxy objects (section 2.3) or attached databases (section 4.2.2). But are they powerful enough to deal with real serious applications users are currently confronted with? Once that a common programming interface, maybe based on the R/S database interface, is available, extensions in this direction are possible.

3.5 Discussion

Apart from having different interfaces to different engines, sophisticated users do not have problems using databases from within R with the packages currently available. Before defining a common interface we should discuss the needs of unsophisticated users. Using relational databases is not that easy. Both `MySQL` (see [1]) and `mSQL` are relatively simple due to their limited functionality. Others like Oracle or Sybase need a detailed knowledge of database systems. In a client-server environment, a user is usually neither allowed to create databases himself nor to delete them. Furthermore, the power of relational databases is based on relations. It is hard to formulate joins in other languages than SQL.

4 Future Directions: The Providers' Perspective

In this section we motivate the idea of a common interface to databases; we look at some of the most successful approaches to database connectivity that have been implemented in other computing environments, such as Microsoft, C/C++, Java, Perl, and Python; and finally we extract the necessary elements that we would need to implement a common database interface.

4.1 A common API as a tool for package development

Package providers who want or need to connect to DBMS probably need a common API more than end-users. Clearly the convenience of a common interface for accessing DBMS is desirable, but it becomes a necessity for package developers that either want or need to provide tools that should work seamlessly when the data resides on a DBMS.

4.2 Examples

We consider only two examples: one that deals with very large datasets, the other example provides an S view of a DBMS.

4.2.1 Sampling as a means to deal with very large data sets

Suppose our data resides in a DBMS (say, Oracle) and we have reasons to believe that a "sensible" sampling scheme could help us answer the questions that we are considering. Then we could write a library to implement various sampling techniques (simple random sampling, stratified, systematic, clustered) with various allocation methods (with/without replacement, allocation proportional to size, etc.) for our DBMS.

In order for such a library to be able to sample data not just from Oracle, but PostgreSQL, Informix, or Microsoft SQL Server, we need to abstract out the details of interfacing to the DBMS.

4.2.2 (Almost) transparent access to DBMS

Suppose we would like to allow users to attach the contents of their DBMS to their search path and use S semantics to access remote tables. It would be very desirable to have these libraries access data independently of the database engine, being it MySQL, Ingres, or IBM's DB/2.

Note on terminology: The S language defines a *database* as a collection of objects, each with an associated name; S unifies all operations that group objects by name under the idea of databases. S also provides semantics for the access and assignment of data to databases (basically the reading and writing of objects). Furthermore, S uses multiple databases for storing and retrieving objects listed in a so-called "search path", that may be modified through calls to `attach()` and `detach()`.

This mechanism is extensible through *user-defined* databases[6]. (For more details see [2], and [3].) To attach a user-defined database to the search path we need to

1. have the user-defined database class extend `database`
2. write methods for the following generic functions:
   - `dbobjects()`
   - `dbread()`
   - `dbwrite()`
   - `dbremove()`
   - `dbexists()` (optional)

Actions to be taken upon attaching and detaching may be coded in functions `.on.attach()` and `dbdetach()`.

Notice the simplicity in attaching any user-defined database: we only need to extend a (virtual) class and supply methods for pre-defined generic functions. We will try to follow this paradigm in the implementation of a common database interface.

```
# "con" is a connection to some DBMS
> is(con, "dbConnection")
[1] T
> db <- attach(con, max.rows=10000, translate=T)
> search()
[1] ".Data"      "vital_test" "splus"      "stat"
[5] "data"       "trellis"    "main"

> ls(pos=2)[sample(1:26, size=5)]
[1] "TRANTABLE"              "tmpApps"
```

[6] This mechanism is currently not defined in R

```
[3] "MYTRANVIEW"              "TRANMAPTABLE"
[5] "AGGREGATE.APP.TRANTABLE"

> assign("fuel.frame", fuel.frame, where = 2)
> exists("fuel.frame", where = db)
[1] T
> remove("fuel.frame", where = 2)
[1] T

> names(CLIENTMAPTABLE)
 [1] "CLIENTID"        "CLIENTNAME"
 [3] "LASTIPADDRESS"   "LASTENTRYROUTER"
 [5] "TIMEZONE"        "OS"
 [7] "CPU"             "PHYSICALMEM"
 [9] "PAGEFILESIZE"    "TOTALDISKSPACE"
[11] "AVAILDISKSPACE"  "LASTUPLOADTIME"
[13] "HOSTNAME"        "SUBNETMASK"
[15] "AGENTTYPE"       "AGENTLICENSES"
[17] "AGENTVERSION"    "LASTGROUPID"

> table(CLIENTMAPTABLE$AGENTTYPE)
  NMCORP  NMCORPI  NMPRO
      28       14     22

> dim(TRANTABLE)
Problem in dbread(..:
  table TRANTABLE with 118411 rows exceeds
  the threshold limit of 10000 max.rows
Use traceback() to see the call stack
```

The following is worth noting:

- We get (almost) transparent access to DBMS
- Familiar S semantics (`get`, `assign`, etc.)
- This is DBMS independent, and
- Provides an extensible mechanism for user-defined database (e.g., `proxyDBMS`) due to a well-defined and simple interface.

The mechanism works well because it abstracts out the details and specifics of the various DBMS — we only need to provide methods for the specified generic functions and extend the `database` class. However, there are some *caveats*:

- The user-defined databases approach is mainly static. Objects (tables) that get removed or created in the DBMS will not appear in the S attached database, unless we

explicitly `synchronize()` it. It would be nice to have a mechanism[7] by which S would get notified of events such as table creation and deletion, among others.

- The S evaluator will look for objects — *even* functions — in the user-defined database (but see the argument `purpose` of `attach()`). That is, the S semantics allow us to either use a user-defined database in the search path or not, but it does not provide finer resolution; for instance, we can not specify that our user-defined database has no language objects.

- The current S semantics could be extended to allow incremental reads and writes. It would be very advantageous to be able to append to an existing object, and to read portions of an object (say, specified through the existing subscripting mechanism) without having to read or write the object in its entirety.

4.3 Approaches to DBMS Connectivity

The client-server architecture in DBMS has been around for more than 20 years. Other languages and environment have faced and solved the very same problems we are facing in the S language community.

At the core of the communications with DBMS there is a DBMS-dependent protocol that is hidden to the applications by a set of library functions. Each DBMS provides its own library for communication, and applications that need to interface with more than one DBMS need to cope with this diversity. How to unify all these diverse interfaces under a common tool-set is the challenge that applications developers have been facing in environments such as C/C++, Java, Perl, and Python, SAS, SPSS, and the S language community.

So it is worthwhile taking a look at how other languages have solved the problems we currently face in connecting to DBMS from R and S.

- Native API's (C/C++).

  All DBMS provide some kind of native C or C++ interface. The interface consists of a library of C/C++ functions that users may call from their applications. These libraries, typically, interface to the database engine through highly specialized and tuned protocols, therefore, they are highly non-portable but applications can achieved unparalleled performance.

  All approaches below are layers on top of these native API's.

- Embedded SQL

  Easy (easier?) to port. One writes applications in languages like, C/C++, Ada, Cobol, Pascal, Fortran, including (embedding) regular SQL statements in the program. A pre-processor or pre-compiler process these files and substitute the SQL instructions with calls to the native API's. Some of this approach advantages are its simplicity (one does need to know SQL, but not the native API), and better portability (there

[7] The Omegahat environment `http://www.omegahat.org` already provides this type of functionality.

are standards that describe how to embed SQL in various languages). The main disadvantage is that, because the code is pre-processed, some flexibility is lost, besides the problems typically associated with pre-processing.

- ODBC

  Defined mainly at the C/C++ level, it provides C/C++ applications an abstraction layer so they can operate uniformly on data stored on DBMS and other database types without having to know the details of the database server. It provides good support under Windows, where the availability of good drivers is not too much of a problem. As mentioned in section 2, there are Unix/Linux drivers too.

  Unlike JDBC, Perl DBI, and Python DB-API, ODBC is not explicitly based on object-oriented programming.

- JDBC (Java Database Connectivity).

  Similar to ODBC, it provides an abstraction layer on top of native API's for Java applications. JDBC is well defined, with better driver availability on most Unix platforms, and it is included in all JDK's (Java Development Kits).

- Perl DBI (Database Independent) module

  Similar in spirit to ODBC and JDBC, for providing a database independent interface to databases, but for Perl applications. There are a variety of drivers, from ODBC to based on native API.

- Python DB-API

  Similar to Perl DBI and JDBC but tailor to python applications.

- Commercial connectivity tools (e.g., `www.merant.com`)

- Three-tier (CORBA, RMI)

All these approaches (except the native API) to database connectivity follow more or less the same implementation to provide a database-independent method based on object-oriented programming techniques.

4.3.1 An Object-oriented Approach

All these approaches separate the connection to the DBMS into a "front-end" and a "back-end". Applications use only the exposed "front-end" or API. The specific DBMS facilities (Oracle, PostgreSQL, etc.) are provided by device "drivers" that get invoked behind the scenes. These are the common features:

- A driver manager

  This loads and initializes the proper driver (database-specific code) and transparently dispatches methods to the driver. (Note that in the case of S, this can be trivial to implement, since the S evaluator dispatches methods automatically for us.)

- A set of classes and functions that define what operations are possible and how they are defined, e.g.:

  - connect/disconnect to the DBMS
  - create and execute statements in the DBMS
  - extract results/output from statements
  - transaction management
  - information (metadata) from database objects
  - error/exception handling

  (some things are left explicitly unspecified, e.g., authorization, and even the query language!).

- Drivers

  Drivers are collection of functions that implement the functionality defined above in the context of specific DBMS.

- Data type mappings

  Mappings between generic DBMS data types and the environment (C/C++, Java, Perl, Python).

- Utilities

  These deal with non-primitive types (dates, currency, ...), styles of identifiers (upper/lower case), etc.

These API's provide the "look-and-feel" of the corresponding environment (C++, Java, Perl, Python, etc).

4.3.2 The Perl DBI Interface

The following figure depicts a view of the Perl DBI implementation. Applications that need to access databases do so through what we call *generic* functions in S. These are calls into the DBI (database independent) functions. The Perl interpreter then automatically delegates these actions to the DBD (database dependent) driver code. In this manner, the application can transparently access any database for which a driver exists.

4.4 Elements for a Common R/S DBMS API

We now identify the major elements required for the implementation of a common interface to databases from R and S. First, the class structure for the API, namely,

- Classes and their inheritance structure
- Set of generic functions

We then need to specify a set of reasonable data type mappings

- Basic data types are "easy"
- Dates/Times/Currency are not hard, but details need to be thought out

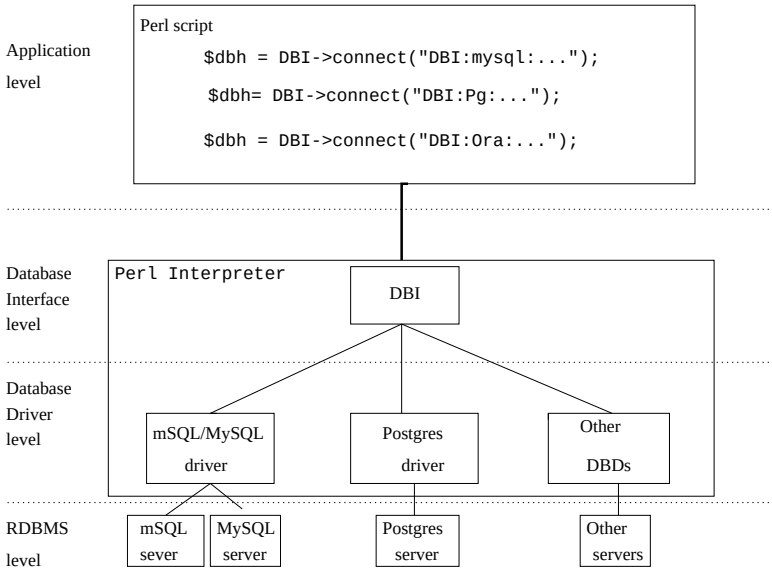

Figure 1: Figure taken from [6].

- Binary data (images, sound, . . .)
- User-defined conversion functions
- Errors/warnings (truncation, loss of precision, . . .)

R/S objects that will mirror the database objects:

- `data.frame` s may be inadequate to represent database objects.
- `raw` type is not available in R, should it?
- how to handle images, sound, etc.

We also need to define the behavior of the interface to handle situations such as long-running queries. Should we allow asynchronous communication? How should R/S be notified when the long-running query has finished? How do we cancel asynchronously invoked queries? How do we insure that the database is properly notified and its resources freed in the event of an application crashing?

The issue of synchronous versus asynchronous communication is very important. Currently many DBMS provide asynchronous communications natively either via *polling* or *call back* functions:

DBMS	Asynchronous communication
Informix	polling (call backs?)
Ingres	call backs
MS SQL Server	call backs
Sybase	call backs
ODBC	polling
Oracle	(threads?)
PostgreSQL	polling (plus listener/notification)
MySQL	threads

Do we need some kind of event notification/handling in R/S? Is the S version 4 mechanism of connections and readers a feasible solution? Should it be implemented in R?

These questions, although critical for communicating with DBMSs, need to be addressed more generally in the context of inter-system communications, since they are pertinent to R/S communications with GUI's, GIS (graphical information systems), other languages (e.g., Java, Perl, Python) spreadsheets (e.g., Microsoft Excel, Gnumeric), and so on.

4.5 Summary

DBMS are much more than simple data repositories. Current R/S facilities already provide some connectivity, but in an uncoordinated fashion. R/S connectivity to DBMS is yet-another application of inter-system communications, and as such, it provides similar benefits and challenges:

- Distributed and parallel computing
- Foreign (remote) references
- Proxy objects
- Asynchronous communications
- Event handling
- Multiple evaluators/Threading?
- Exceptions

A number of issues that come up in the context of DBMS communications also come up when communicating with GUI's, GIS, other languages, etc. In addition to the above benefits and challenges, we need to address the following issues:

- Should R support user-defined databases?
- Should the current semantics of user-defined databases in S be extended to include asynchronous communications, incremental read/write, etc?
- Do we need a `raw` type in R?

References

[1] David Axmark, Michael Widenius, Jeremy Cole, and Paul DuBois. *MySQL Reference Manual*. http://www.mysql.com/documentation/mysql, 2001.

[2] John M. Chambers. Data management in S. Technical report, Bell Labs, Lucent Technologies, http://stat.bell-labs.com/stat/doc, 1991.

[3] John M. Chambers. Database classes. Technical report, Bell Labs, Lucent Technologies, http://stat.bell-labs.com/stat/Sbook, 1998.

[4] Peter Dalgaard. The R-Tcl/Tk interface. In *Proceedings of the Distributed Statistical Computing 2001 Workshop*, http://www.ci.tuwien.ac.at/Conferences/DSC-2001, 2001. Vienna University of Technology.

[5] Alligator Descartes and Tim Bunce. *Programming the Perl DBI*. O'Reilly, 2000.

[6] Paul DuBois. *MySQL*. New Riders, 2000.

[7] Jon Ellis, Linda Ho, and Maydene Fisher. *JDBC 3.0 Specification*. Sun Microsystems, Inc, http://java.sun.com/Download4, 2000.

[8] Duncan Temple Lang. Embedding S in other languages and environments. In *Proceedings of the Distributed Statistical Computing 2001 Workshop*, http://www.ci.tuwien.ac.at/Conferences/DSC-2001, 2001. Vienna University of Technology.

[9] Microsoft Inc, http://www.microsoft.com/data/odbc/. *Microsoft ODBC*, 2001.

[10] Erich Neuwirth and Thomas Baier. Embedding R in standard software, and the other way around. In *Proceedings of the Distributed Statistical Computing 2001 Workshop*, http://www.ci.tuwien.ac.at/Conferences/DSC-2001, 2001. Vienna University of Technology.

[11] George Reese. *Database Programming with JDBC and Java*. O'Reilly, second edition, 2000.

[12] B. D. Ripley and R. M. Ripley. Applications of R clients and servers. In *Proceedings of the Distributed Statistical Computing 2001 Workshop*, http://www.ci.tuwien.ac.at/Conferences/DSC-2001, 2001. Vienna University of Technology.

[13] Brian D. Ripley. Using databases with R. *R News*, 1(1):18–20, January 2001.

[14] R Development Core Team. *R Data Import/Export*. http://www.r-project.org, 2001.