1 Accessing a Database from R

We have noted already that SQL has limited numerical and statistical features. For example, it has no least squares fitting procedures, and to find quantiles requires a sophisticated query. (Celko discusses the pros and cons of more than eight different advanced queries to find a median [?].) Not only are basic statistical functions missing from SQL, but in many cases the numerical algorithms used in the basic aggregate functions are not implemented to safeguard numerical accuracy. Also, the wide range of data types may have drawbacks when it comes to performing arithmetic calculations across a row, as some of the conversions from one numeric type to another may produce unexpected truncation and rounding. For these reasons, it may be desireable or even necessary to perform a statistical analysis in a statistical package rather than in the database. One way to do this, is to extract the data from the database and import it into statistical software.

The statistical software may either reside on the server-side, i.e. on the machine which hosts the database, or it may reside on the client-side, i.e. the user's machine. The DBI package in R provides a uniform, client-side interface to different database management systems, such as MySQL, PostgreSQL, and Oracle. The basic model breaks the interface between the client and the server into three main elements: the *driver* facilitates the communication between the R session and a particular type of database management system (e.g. MySQL); the *connection* encapsulates the actual connection (with the aid of the driver) to a particular database management system and carries out the requested queries; and the *result* which tracks the status of a query, such as the number of rows that have been fetched and whether or not the query has completed.

The DBI package provides a general interface to a database management system. Additional packages that handle the specifics for particular database management systems are required. For example, the RMySQL package extends the DBI package to provide a MySQL driver and the detailed inner workings for the generic functions to connect, disconnect, and submit and track queries. The RMySQL package uses client-side software provided by the database vendor to manage the connection, send queries, and fetch results. The R code the user writes to establish a MySQL driver, connect to a MySQL database, and request results is the same code for all SQL-standard database managers.

We provide a simple example here of how to extract data from a MySQL database in an R session. The first step: load a driver for a MySQL-type database:

```
 drv = dbDriver("MySQL")
```

The next step is to make a connection to the database management server of interest. This connection stays alive for as long as you want it. For some types of database management systems, such as MySQL, the user can establish multiple connections: each one to a different database or different server. Below, the user s133cs establishes a connection, called *con*, to the database named BaseballDataBank on the host statdocs.berkeley.edu. Since the database is not password protected, the user need not provide a password to gain access to it.

```
con = dbConnect(drv, user="s133cs", dbname="BaseballDataBank",
         host="statdocs.berkeley.edu")
```

Once the connection is established, queries can be sent to the database. Some queries are sent via R functions. For example, the following call to the dbListTables function submits a SHOW TABLES query that gets remotely executed on the database server. It returns the names of the tables in the BaseballDataBank database.

```
dbListTables(con)
```

As another example, the dbReadTable function performs simple SELECT queries that mimics the R counterpart 'get.' That is, dbReadTable imports the Allstar table from the database into R as a data frame, using the attribute PlayerID as the row.names for the data frame.

```
dbReadTable(con, "Allstar", row.names = "PlayerID")
```

Other RMySQL functions are dbWriteTable, dbExistsTable, and dbRemoveTable, which are equivalent to the R functions 'assign', 'exists', and 'remove', respectively.

Other queries can be executed by supplying the SQL statement. For example, to perform a simple aggregate query, there is no need to pull a database table into R and apply an R function to the data frame. Instead, we issue a select statement and retrieve the results table as a data frame. Below is an example where we obtain the number of tuples in the Allstar table of BaseballDataBank.

```
dbGetQuery(con,"SELECT COUNT(*) FROM Allstar;")
```

When the result table is huge, we may not want to bring it into R in its entirety, but instead fetch the tuples in batches, possibly reducing the batches to simple summaries before requesting the next batch. We provide a detailed example of this approach in Section **??**. Instead of dbGetQuery, we use dbSendQuery to fetch results in batches. The DBI package provides functions to keep track of whether the statement produces output, how many rows were affected by the operation, how many rows have been fetched (if statement is a query), and whether there are more rows to fetch.

In the example below, rather than using dbReadTable to pull over the entire TCPConnections table, the dbSendQuery function is used to send the query to the database without retrieving the results. Then, the fetch function pulls over tuples in blocks. In this example, the first 500 tuples are retrieved, then the next 200, after which we determine that there are more results to be fetched (dbHasCompleted) and clear the results object (dbClearResult) without bringing over any more tuples from the SQL server.

```
rs = dbSendQuery(con2, "SELECT * FROM TCPConnections;")

firstBatch =  fetch(rs, n = 500)
secondBatch =  fetch(rs, n = 200)

dbHasCompleted(rs)

dbClearResult(rs)
```

In addition, the $n = -1$ assignment for the parameter specifies that all remaining tuples are to fetched. The fetch function converts each attribute in the result set to the corresponding type in R. In addition, dbListResults(con) gives a list of all currently active result set objects for the connection con, and dbGetRowCount(rs) provides a status of the number of rows that have been fetched in the query. When finished, we free up resources by disconnecting and unloading the driver:

```
dbDisconnect(con)

dbUnloadDriver(drv)
```