# 3

# R and Databases

While many tasks that used to be performed using relational databases can be easily implemented in R, there are some situations where using the power of a relational database nicely complements the capabilities of R. One obvious example are situations where the data to be used is stored in a relational database. Relational databases can also be used to make working with very large datasets easier.

The topic of administration of a database is beyond the scope of this book, and the assumption will always be made that you have access to a running database, and that enough permissions have been granted to perform the necessary database operations.

There are two principal ways to connect with databases in R. The first uses the `ODBC` (Open DataBase Connectivity) facility available on many computers. The second uses the `DBI` package of R along with a specialized package for the particular database needed to be accessed. If there is a specialized package available for your database, you may find that the corresponding `DBI`-based package may give better performance than the `ODBC` approach. On the other hand, if you are using a database for which a specialized package is not available, using `ODBC` may be your only option.

## 3.1 A Brief Guide to SQL

### 3.1.1 Navigation Commands

Since a single server may hold more than one database, each with potentially many tables, and since each table can contain many columns (variables), it may be useful to examine exactly what's available in a database before starting to work with it. Often there are graphical clients available to communicate with databases that will present this information in a convenient form, but R can also be used to create data frames containing this information. The table below shows some common tasks and the SQL statements to execute

them; when used with `dbGetQuery` they will each return a data frame with the requested information. In the table below, keywords are shown in uppercase; the terms in lowercase would be replaced by those specific to your task. When

| Task | SQL query |
|---|---|
| Find names of available databases | `SHOW DATABASES` |
| Find names of tables in a database | `SHOW TABLES IN database` |
| Find names of columns in a table | `SHOW COLUMNS IN table` |
| Find the types of columns in a table | `DESCRIBE table` |
| Change the default database | `USE database` |

**Table 3.1.** Basic SQL commands

using command-line clients, each SQL statement must end in a semicolon, but the semicolon is not required when using the `RMySQL` interface. Keywords will always be recognized regardless of case, but depending on the version of MySQL that the server is using, database, table, and column names may or may not be case-sensitive.

### 3.1.2 Basics of SQL

The first step to understanding SQL is to realize that, unlike R, it is not a programming language; operations in SQL are performed using individual queries without loops or control statements. The most important SQL command is `SELECT`. Since queries are performed using single statements, the syntax of the `SELECT` command can be quite daunting:

```
SELECT columns or computations
     FROM table
     WHERE condition
     GROUP BY columns
     HAVING condition
     ORDER BY column  [ASC | DESC]
     LIMIT offset,count;
```

Fortunately, most of the clauses in the `SELECT` statement are optional. In fact, many queries will simply retrieve all of the data in a particular table through the following command:

```
SELECT * FROM tablename;
```

The asterisk (`*`) means "all the columns in the table". Alternatively, a comma-separated list of variables or expressions can be supplied:

```
SELECT var1,var2,var2/var1 from tablename;
```

will return three columns corresponding to `var1`, `var2`, and the computed value of their ratio. A useful operator in `SQL` is the `AS` operator, which can be used to change the name of a column in the result set. In the previous example, if we wanted the name of the third column to be "`ratio`" we could use the `AS` command:

```
SELECT var1,var2,var2/var1 AS ratio FROM tablename;
```

In fact, the use of the word `AS` is optional; the new column name can simply follow the old one. In these examples, I will include the `AS` keyword, as it makes the query more readable. This same technique can be used to refer to tables through alternative names as well.

To limit the rows which are returned, the `WHERE` clause can be used. Most common operators can be used to define expressions for the `WHERE` clause, along with the keywords `AND` and `OR`. For example, to extract all columns for the rows of a table where `var1` is greater than 10 and `var2` is less than `var1`, we could use

```
SELECT * FROM tablename WHERE var1 > 10 AND var2 < var1;
```

One limitation of the `WHERE` clause is that it cannot access variables that were created in the `SELECT` statement; the `HAVING` clause must be used in those cases. So to find cases where the computed ratio is greater than 10, we could use a statement like this:

```
SELECT var1,var2,var2/var1 AS ratio
   FROM tablename HAVING ratio > 10;
```

Notice the similarity between these simple queries and the `subset` function (Section 6.8).

Two operators in SQL are especially useful for character variables. The `LIKE` operator allows the use of "`%`" to represent zero or more of any character, and "`_`" to represent exactly one character. The `RLIKE` operator allows the use of regular expressions for character comparisons (see Section 7.4).

### 3.1.3 Aggregation

The `GROUP BY` clause, in conjunction with some SQL-provided aggregation functions, can be useful if you wish to produce a table of counts or a data summary from a database, without bringing all of the data into R. Some of the common aggregation functions available in SQL are summarized in Table 3.2. For example, suppose we wanted to create a table of means for a variable `x`, from a database table named `table`, broken down by a categorical variable called `type`. We could create a table with the value of `type` and `mean` with the following statement:

```
SELECT type,AVG(x) AS mean FROM table GROUP BY type;
```

| Task | SQL aggregation function |
|---|---|
| Count numbers of occurrences | `COUNT()` |
| Find the mean | `AVG()` |
| Find minimum | `MIN()` |
| Find maximum | `MAX()` |
| Find variance | `VAR_SAMP()` |
| Find standard deviation | `STDDEV_SAMP()` |

**Table 3.2.** Basic SQL aggregation commands

Remember to include the grouping variable in the list of selected variables, as SQL will not do this automatically. Since `mean` is a calculated variable in this example, you would need to use a `HAVING` clause to limit the observations that were returned based on the value of `mean`.

Since the number of observations for any column in a particular table will always be the same, it is common practice to use an asterisk (`*`) as an argument to the `COUNT` aggregation function. To create a table of counts by `type` in the previous example, we could use

```
SELECT type,COUNT(*) FROM table GROUP BY type;
```

To group by more than one variable, use a comma-separated list as an argument to the `GROUP BY` clause.

Multiple aggregated statistics can easily be output in a single query. Suppose we wanted to count the number of observations for each `type`, along with the mean and standard deviation of the `x` column. The following command could be used:

```
SELECT type,COUNT(*),AVG(x) AS mean,STDDEV_SAMP(x) AS std
   FROM accounts GROUP BY type;
```

### 3.1.4 Joining Two Databases

One of the strengths of database servers is that they can effectively join together multiple database tables, based on common values of columns within the tables. Of course, the same capability is available within R through the `merge` function (see Section 9.6), but it may be more efficient to use the database server for merging.

The most common way of joining two tables is through an inner join; only those observations that have common values of the variable used for the merge will be retained in the output table. (This is also the default behavior of the `merge` function.) For example, suppose we have a table called `children` with columns `id`, `family_id`, and `height` and `weight`, and a second table called `mothers`, with columns `id`, `family_id`, and `income`. We would like a table with the `height` and `weight` of the children, along with the income of the mothers. The following SQL statement will return the table:

```
SELECT height,weight,income FROM children
   INNER JOIN mothers USING(family_id);
```

The variable in the `USING` expression (`family_id` in this example) is known as a key or sometimes a foreign key. If the two tables being joined have only one variable in common, the `INNER JOIN` can be replaced with a `NATURAL JOIN`, and the `USING` expression can be omitted.

Now suppose we wish to produce a table with both the children's `id` and the mother's `id`. Since there are variables called `id` in both data tables, we need to distinguish between them by preceding the column name with the table name and a period. In this example, we could use a query like this:

```
SELECT children.id,mothers.id,height,weight,income
   FROM children INNER JOIN mothers USING(family_id);
```

The `AS` operator can be used to make it easier to refer to multiple tables, as well as renaming the columns:

```
SELECT c.id as kidid,m.id as momid,height,weight,income
   FROM children AS c
   INNER JOIN mothers AS m USING(family_id);
```

### 3.1.5 Subqueries

Continuing with the current example, consider the task of tabulating the family size (i.e. the number of children with the same `family_id`) for all the families in the database. It's easy to create a table that has the counts for each value of `family_id`:

```
SELECT family_id,COUNT(*) AS ct FROM children
   GROUP BY family_id;
```

How can we then count how many of each family size was found? One way would be to create a temporary table containing the ids and sizes, and querying that table, but often the permission to create new tables on a server is not available. The alternative is to use subqueries. In SQL, a subquery is a query surrounded by parentheses, which can be treated just like any other table. One restriction of subqueries is that all subquery tables must be given an alias (through the `AS` operator), even if you won't be directly referring to the table. We can produce the table of family sizes with the following query:

```
SELECT ct,COUNT(*) as n
  FROM (SELECT COUNT(*) AS ct FROM children
        GROUP BY family_id) AS x
  GROUP BY ct;
```

Subqueries are also useful when the timing of database operations makes a query impossible for the database to understand. Let's say we wanted all the available information about the tallest child in the database. One obvious possibility is to perform the following query:

```
SELECT * FROM children WHERE height = MAX(height);
```

Depending on the database you use, you might get an empty set, or a syntax error. To get around the problem, we can create a table with only the maximum height, and then use it in a subquery:

```
SELECT * FROM children
   WHERE height = (SELECT MAX(height) as height from children);
```

### 3.1.6 Modifying Database Records

To change the values of selected records in a database, the `UPDATE` command can be used. The format of the `UPDATE` statement is

```
UPDATE table SET var=value
   WHERE condition
   LIMIT n;
```

To change more than one variable's value, the `var=value` specification can be replaced with a comma-separated list of variable/value pairs. The `WHERE` and `LIMIT` specifications are optional. If a `LIMIT` specification is provided, only that many records will be considered for updating, even if some of the chosen records will not actually be modified. For example, to change the height and weight for a subject with a particular id, we could use a statement like

```
UPDATE children SET weight=100,height=55
   WHERE id = 12345;
```

To completely remove a record, the `DELETE` statement can be used. The basic syntax is as follows:

```
DELETE FROM table
   WHERE condition
   LIMIT n;
```

Without a `WHERE` clause, all of the records of the database table will be removed, so this statement should be used with caution. If a `LIMIT` specification is provided, it will be based on observations matching the condition of the `WHERE` clause, if one is specified.

Finally, to completely remove an entire table or database, the `DROP` statement can be used, for example

```
DROP TABLE tablename;
```

or

```
DROP DATABASE dbname;
```

When using the `DROP` command, an error will be reported if the table or database to be dropped does not exist. To avoid this, the `IF EXISTS` clause can be added to the `DROP` statement, as in

```
DROP DATABASE IF EXISTS dbname;
```

Notice that these commands take effect on the database as soon as they are issued, so it's a good idea to have a backup of the data in the database before using these commands.

## 3.2 ODBC

The ODBC (Open DataBase Connectivity) facility allows access to a variety of databases through a common interface. In R, the RODBC package, available from CRAN, is used to access this capability. ODBC was originally developed on Windows, and the widest variety of ODBC connectors will be available on that platform. However, both Linux and Mac OS X also provide database connectivity through ODBC. If you need to use a database in R that is not directly supported, RODBC will probably be your best choice, as many database manufacturers provide ODBC connectors for their products.

The first step in using RODBC is to set up a DSN or data source name. In order to do this, you need to know the name that your computer uses for a particular data source. On Windows, the ODBC Source Administrator (accessed through Control Panel → Administrative Tools → Data Sources(ODBC)) is used to establish DSNs. Under the "Drivers" tab, you can see what connectors are available on your computer, and the name that is used to access them. If you install additional connectors, you should see them listed here. You can use this name, providing additional connection details each time you create a connection, or you can create a new DSN to automate the process. To create a new DSN, click on the "Add" button under the User DSN tab, choose an appropriate driver from the pop-up window, and click "Finish". At this point a dialog specific to the database you're using will appear, and you can fill in the required information to create the DSN. Make sure to note the name that you use for the data source name, since that is how the ODBC connection is specified.

Under Mac OS X, the ODBC Administrator (which can be found in the /Applications/Utilities folder) performs a similar function. You can view available drivers in the "Drivers" tab, or choose the User DSN tab, and click "Add" to create a new DSN; after choosing a driver, you can configure it using keyword/value pairs appropriate for the particular database you are using.

To use the RODBC package on a Linux system, the unixodbc libraries must first be installed. Most linux distributions will make this very easy. The configuration of UNIXODBC is controlled by two files: odbcinst.ini and odbc.ini. The first file contains the available ODBC drivers, and the second file is used to define additional DSNs, if desired. For example, the following is an odbc.ini file which defines a DSN called myodbc using the MySQL ODBC driver:

```
[myodbc]
Driver       = MySQL
Description  = MySQL ODBC 2.50 Driver DSN
Server       = localhost
Port         = 3306
User         = user
Password     = password
Database     = test
```

The name in square brackets (`myodbc` in this case) is the `DSN` that is being defined; multiple `DSN`s can be defined in a single file by starting a new section with the `DSN` in brackets. In order to use a driver, it must be defined in the `odbcinst.ini` file. The specific keywords in the file will depend on the specific connector being used.

By default on most systems, the two configuration files for `UNIXODBC` are in the `/etc` directory. To specify a different location for `odbc.ini`, set the environmental variable `ODBCINI` to the fully-qualified filename of the file; to specify a different location for `odbcinst.ini`, set the environmental variable `ODBCSYSINI` to the **directory** in which `odbcinst.ini` can be found.

## 3.3 Using the `RODBC` Package

After loading the `RODBC` package, if you've configured a `DSN` that provides all the necessary information to connect and access your database, you can create a connection by simply passing the `DSN` to the `odbcConnect` function. Suppose we have a `DSN` named `myodbc` to connect to a `MySQL` database, and we have provided the server, username, password, and database in the `DSN` definition. Then we can create a connection through `RODBC` as follows:

```
> library(RODBC)
> con = odbcConnect('myodbc')
```

Additional keywords defining the connection can be provided in the `DSN` argument by separating `keyword=value` pairs with semicolons. For example, if a `DSN` was created without specifying a required password, the database could be accessed as follows:

```
> con = odbcConnect('myodbc;password=xxxxx')
```

Other possible keywords depend on the particular data source. For `MySQL` these keywords include `server`, `user`, `password`, `port`, and `database`; for `PostgreSQL`, substitute `username` for `user`.

Once you've got a connection to the `ODBC` source, the `sqlQuery` function allows any valid `SQL` query to be sent to the connection. This will be the case even if `SQL` is not the native language of the underlying database. Passed only a connection and a query, `sqlQuery` will return a data frame containing the

entire result of the query. The `max=` argument to `sqlQuery` will limit the number of rows returned, and can be followed by repeated calls to `sqlGetResults` (also using appropriate `max=` arguments) to process a query in smaller pieces.

To prevent unnecessary resource use, the `odbcClose` function should be passed any `ODBC` connection objects when they are no longer needed.

## 3.4 The `DBI` Package

One of the most popular databases used with R is MySQL (`http://mysql.com`). This freely available database runs on a variety of platforms and is relatively easy to configure and operate.

In the following sections, we'll look at the `RMySQL` package as an example of using the `DBI` package.

## 3.5 Accessing a MySQL Database

The first step in accessing a MySQL database is loading the MySQL package. This package will automatically load the required `DBI` package, which provides a common interface across different databases. Next, the MySQL driver is loaded via the `dbDriver` function, so that the `DBI` interface will know what type of database it's communicating with:

```
> library(RMySQL)
> drv = dbDriver("MySQL")
```

Now, the specifics of the database connection can be provided through the `dbConnect` function. These include the database name, the database username and password, and the host on which the database is running. If the database is running on the same machine as your R session, the hostname can be omitted. For example, to access a database called "`test`", via a user name of "`sqluser`" and password of "`secret`" on the host "`sql.company.com`", the following call to `dbConnect` could be used:

```
> con = dbConnect(drv,dbname='test',user='sqluser',
+                 password='secret',host='sql.company.com')
```

The calls to `dbDriver` and `dbConnect` need only be made once for an entire session. Note that the `dbname` passed to `dbConnect` might represent a collection of many tables; the specific table to be used will be specified in the queries that are sent to the database.

You can close an unused `DBI` connection by passing the connection object to `dbDisconnect`.

## 3.6 Performing Queries

SQL queries make requests for some or all of the variables in one or more database tables, so a natural way to package these results within R is in a data frame. In most cases, a single call to `dbGetQuery` can be used to send a query to the database, and have the resulting table returned as a data frame. For example, suppose that we have connected to the database "`test`" as described in the previous section, and we wish to extract all of the observations in a table called "`mydata`". After the appropriate calls to `dbDriver` and `dbConnect`, we could retrieve the data with the following command:

```
> mydata = dbGetQuery(con,'select * from mydata')
```

Any valid SQL query can be passed to a database by this method.

In the case where data needs to be processed in pieces, the `dbSendQuery` function can be used to initiate the query, and the `fetch` function can be passed the result from `dbSendQuery` to sequentially access the results of the query. Once all the required data is extracted using `fetch`, the result from `dbSendQuery` should be passed to the `dbClearResult` function to insure that the next query will be properly processed. (When using `dbGetQuery` there is no need to call an additional function at the end of the query.) Note that by default, the `fetch` function will return 500 records at a time; this can be overwritten with the `n=` argument, using a value of -1 to indicate all of the available records, or an integer to specify the number of records desired.

## 3.7 Normalized Tables

The principle of normalization is central to database design. The goal of normalization is to eliminate redundancy in the information stored in the database tables. To achieve this goal, what might be a single data frame in R might be broken into several tables in a database. For example, suppose we are working with a database containing information about the parts required to produce a product. If we stored the part name, supplier's name, and the price of the part all in one database, we would have many records with identical information about suppliers. In a properly normalized database, there would be two tables; one with part names and prices and an id representing the supplier of the part. This id, known as a key or foreign key, would also be found exactly once in a second table containing supplier information. Suppose the first table is called `parts` with columns `name`, `price`, and `supplierid`, and the second table is called `suppliers`, with columns `supplierid` and `name`. Our goal is to create a data frame with the part name and price along with the name of the supplier. An appropriate query to retrieve the table we want into a data frame could be written as

```
> result = dbGetQuery(con,'SELECT parts.name,parts.description,
+                          supplier.name AS supplier
+                          FROM parts INNER JOIN
+                          suppliers USING(supplierid)')
```

Using the database to merge the tables makes sense if you're familiar with SQL, and especially if the tables you're working with are very large. However, the tables could also be retrieved in their entirety, and the merging performed in R:

```
> parts = dbGetQuery(con,'SELECT * FROM parts')
> suppliers = dbGetQuery(con,'SELECT * FROM suppliers')
> result = merge(parts,suppliers,by='supplierid')
```

This simplistic solution, while workable, ignores the motivation behind the initial normalization of the database tables, namely, to avoid redundancy. The supplier name variable is being stored as a character variable, whose value is repeated in the `result` data frame for each observation from the same supplier. A more efficient solution is to note the similarity between the `suppliers` table and the idea of a factor in R. The `supplierid`s represent the levels of a factor, and the `name`s represent the labels. Thus, we can create a data frame storing the suppliers as a factor with code like this:

```
> parts = dbGetQuery(con,'SELECT * FROM parts')
> suppliers = dbGetQuery(con,'SELECT * FROM suppliers')
> result = data.frame(name=parts$name,price=parts$price,
+                          supplier=factor(parts$supplierid,
+                          levels=suppliers$supplierid,
+                          labels=suppliers$name))
```

Since the `data.frame` function automatically converts character variables to factors, both `name` and `supplier` will be stored as factors.

## 3.8 Getting Data into MySQL

If your data is already in an R object, it can be easily transfered to a database using the `dbWriteTable` function, which accepts the same sort of connection object that `dbGetQuery` uses. By using the `append=TRUE` argument to `dbWriteTable`, a large database table can be built using smaller pieces.

If it is desired to create a table directly from raw data, it is first necessary to describe the nature of each column in the table with the `CREATE TABLE` statement. For example, one way of creating a table called `mydata` to hold columns `name` (a character variable) and `number` (a floating point value) would be to issue a statement like:

```
CREATE TABLE mydata (name text, number double);
```

This statement could be submitted to MySQL by, for example, passing it to `dbGetQuery` (although it will not return any value). To make generating statements like this easier, the `dbBuildTableDefinition` function can be used; it will generate appropriate statements to create a database suitable to hold an R data frame. Following the current example, we could generate the `CREATE TABLE` statements in R as follows:

```
> x = data.frame(name='',number=0.)
> cat(dbBuildTableDefinition(dbDriver('MySQL'),
+                            'mydata',x),"\n")
CREATE TABLE mydata
( row_names text,
        name text,
        number double
)
```

To suppress the `row_names` column, the `row.names=FALSE` argument can be used. The output from `dbBuildTableDefinition` can be passed directly to `dbGetQuery` to create the table in the database. If you wish to create a table with identical specifications to an existing table, the `LIKE` clause can be used in the `CREATE TABLE` statement, as in

```
CREATE TABLE newtable LIKE oldtable;
```

To get an understanding of how existing tables are stored in the database, the `DESCRIBE table` statement can be used.

Once the table has been created, the actual data needs to be entered. The SQL `INSERT` command can be used to add one or more observations to a database table. When the columns defined by the `CREATE TABLE` command are being entered in the order they are stored in the database table, all that is required is the `VALUES` keyword:

```
INSERT INTO mydata VALUES('fred',7);
```

If the values are to be entered in an order different from how they are stored in the database table, a parenthetical comma-separated list describing the order that will be used needs to be provided before the `VALUES` keyword. So to add an observation only specifying the `number` value before the `name` value, we could use the following SQL command:

```
INSERT INTO mydata (number,name) values(7,'fred');
```

To add additional observations, additional parenthesized comma-separated lists, themselves separated by commas, can be added at the end of the `INSERT` command. The following command adds two new observations to the `mydata` table:

```
INSERT INTO mydata VALUES('tim',12),('sue',9);
```

Generally, however, it will be advantageous to insert all of the data into the database in a single database call, either through an external program or through the `LOAD DATA` command. With MySQL, the `mysqlimport` shell command can be used to read whole files of data into a database table. Among its arguments are `--local`, which specifies that the data is local, and not on the server, `--delete`, which insures that the contents of any current table with the same name are removed before creating the new table, and `--fields-terminated-by=` and `--lines-terminated-by=` to provide the field and line terminators, respectively. In addition to these optional arguments, the `-u username` option, to provide the MySQL username, the `-h hostname` option to provide the name of the machine on which the MySQL server is running, and the `-p` option, to tell the server to prompt for a password, may be required to establish a database connection. In addition, since the MySQL server won't read header lines, the `--ignore-lines=1` argument can be used to skip a header line.

For example, to read a comma-separated text file called `mydata.txt` into a mysql database called `test`, the following shell command could be entered in a terminal window:

```
mysqlimport -u sqluser -p --delete --local \
            --fields-terminated-by=',' test mydata.txt
```

Notice that `mysqlimport` determines the table name by removing any suffix from the name of the file containing the data (`mydata.txt` in this example). As with the `LOAD DATA` command, the table to hold the data must be created before `mysqlimport` can be used.

The same operation can be performed by sending MySQL statements to the server. Assuming an appropriate connection object has been obtained, we could load data from the `mydata.txt` file into the database with the following call to `dbGetQuery`:

```
> dbGetQuery(con,"LOAD DATA INFILE 'mydata.txt'\
+             INTO TABLE mydata\  FIELDS TERMINATED BY ','")
```

Once the data is loaded into the database, the `SELECT` statement can be used to create subsets of the data which will be manageable inside of R.

## 3.9 More Complex Aggregations

The `dbApply` function can be used to apply a user-specified R function to groups of data extracted from a database. To use `dbApply`, first create a result set object through a call to `dbSendQuery`, using the `ORDER BY` clause to insure that the data will be brought into R in the appropriate order. The result set object can then be passed to `dbApply`, along with an `INDEX=` argument to specify the grouping variable, and a `FUN=` argument, to specify the function to be applied to each group. This function must accept two arguments: the first

is the data frame consisting of the requested data for a given group, and the second is the value of the grouping variable. For example, suppose we have a database table called `cordata` with columns `group`, `x`, and `y`, and we wish to find the correlation between `x` and `y`, broken down by groups. First, we use `dbSendQuery` to create the result set object:

```
> res1 = dbSendQuery(con,
+            'SELECT group,x,y FROM cordata ORDER BY group')
```

Now we can pass this result set object to `dbApply` to obtain the result:

```
> correlations = dbApply(res1,INDEX='group',
+                        FUN=function(df,group)cor(df$x,df$y))
```

The return value from `dbApply`, `correlations`, will be a list of correlations whose names represent the levels of the `group` variable.

If the `dbApply` function is not available for a particular database, or if more control is required over the aggregation, the following function shows an alternative means of applying a function to subsets of the data:

```
mydbapply = function(con,table,groupv,otherv,fun){
    query = paste('select ',groupv,' from ',table,
                  ' group by ',groupv,sep='')
    queryresult = dbGetQuery(con,query)
    answer = list()
    k = 1
    varlist = paste(c(groupv,otherv),collapse=',')
    for(gg in queryresult[[groupv]]){
        qry = paste('select ',varlist,' from ',table,'
                    where ', groupv,' = "',gg,'"',sep='')
        qryresult = dbGetQuery(con,qry)
        answer[[k]] = fun(qryresult)
        names(answer)[k] = as.character(gg)
        k = k + 1
     }
    return(answer)
}
```

The arguments to `mydbapply` are `con`, an active database connection object, `groupv`, a character string representing the database column to be used for grouping, `otherv`, a character vector containing the names of other database columns that need to be extracted from the database, and `fun`, the function that will operate on the data frame containing the grouping and other variables. The example of the previous section could be executed using `mydbapply` as

```
> correlations = mydbapply(con,'cordata','group',c('x','y'),
                           function(df)cor(df$x,df$y))
```