Time Series Database Interface (TSdbi) Guide and Illustrations

Paul D. Gilbert
September 1, 2013

Contents

1 Introduction

This vignette illustrates the various *R* (**?**) *TSdbi* packages using time series data from several sources. The main purpose of *TSdbi* is to provide a common *API*, so simplicity of changing the data source is a primary feature. The vignette also illustrates some simple time series manipulation and plotting using packages *tframe* and *tfplot*.

To generate this vignette requires most of the *TS** packages, but users will only need one, or a few of these packages. For example, a time series database can be built with several SQL database backends, but typically only one would be used. On the other hand, several packages pull data from various Internet sources, so several of these might be used to accommodate the various specifics of the sources.

If package *TSdata* is installed on your system, it should be possible to view the pdf version of this guide with *vignette("Guide", package="TSdata")*. Otherwise, consider getting the pdf file directly from CRAN at `http://cran.at.r-project.org/web/packages/TSdata/index.html` Many parts of the vignette can be run by loading the appropriate packages, but some examples use data that has been loaded into a larger database and it will not be possible to reproduce those examples without the underlying database.

Section 2 of this vignette illustrates the mechanism for connecting to a data source. The connection contains all the source specific information so, once the connection is established, the syntax for retrieving data is similar for different sources. The section also illustrates packages that pull data from Internet sources. This currently includes *TSgetSymbol*, *TShistQuote*, *TSjson*, *TSxls*, and *TSzip*, but others are in progress. These packages only get data, they do not support writing data to the database. Finally, the section also illustrates the flexibility to return different types of time series objects.

Section 3 illustrates the SQL packages *TSPostgreSQL*, *TSMySQL*, *TSSQLite*, and *TSodbc*. These use a very standard SQL table structure and syntax, so it should be possible to use other SQL backends. The package *TSOracle* is available on R-forge at `http://tsdbi.r-forge.r-project.org/` but I currently do not have a server in place to test it properly. (If anyone would be interested in doing this, please contact me, pgilbert.ttv9z@ncf.ca.) These packages get data and also support writing data to the database. See Appendix "B" (section 9) and the vignette in package *TSdbi* for more explanation of the underlying database tables, and for bulk loading of data into a database. The section illustrates writing artificial data to the database.

Section 4 provides additional examples of *TSdbi* functionality and of using and graphing series. Subsection 4.1 illustrates fetching data from the web and loading it into a local database. Subsection **??** provides examples of how one might use a more extensively populated database that has been previously constructed.

Section 5 illustrates package *TScompare* for comparing time series databases, and Section 6 illustrates the use of realtime vintages of data.

Section 7 illustrates some functions for exporting time series data from *R*

into .xls and .csv files. While regular *R* users may not be too interested in this, it can be useful for colleagues that are not yet regular users. The section is fairly self contained. (But will give non-users some exposure to *R*, so they may decide they do not actually need to export the data.)

Appendix "A" provides connection details specific to the different database sources, Appendix "B" provides more details about the structure of SQL databases and, finally, Appendix "C" provides some examples of SQL queries that may be useful for database maintenance.

Many of the *TS** packages are wrappers of other packages. The purpose is to provide a common API for interfacing with time series databases, and an easy mechanism to specify the type of time series object that should be returned, for example, a *ts* object or a *zoo* (**?**) object. One consequence of providing a common interface is that special strengths of some of the underlying packages cannot always be used. If you really need some of these features then you may need to go directly to the underlying package. However, if you limit your reliance on these underlying features then you will be able to move from one data source to another much more easily.

Loading a *TS** package will also load required packages *TSdbi*, *DBI* (**?**), *methods*, *tframePlus*, *zoo*, and then any underlying package that the specific *TS** package uses.

2 Simple Examples of Time Series Data from the Internet

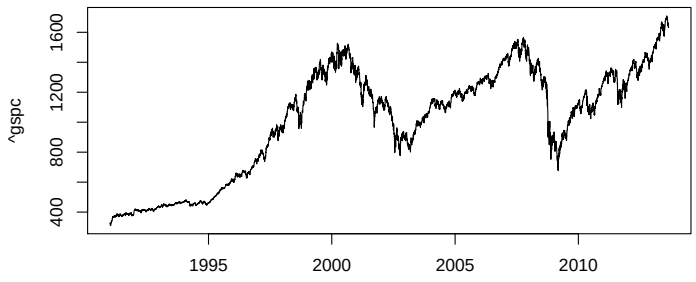
This section uses packages to pull data from the Internet. The general syntax of a connection is illustrated, and some simple calculations and graphs are done to demonstrate how the data might be used. However, the purpose of the *TS** packages is to provide a common interface, not to do all time series calculations and graphics. Once you have the data, you should be able to use whatever other *R* packages you like for your calculations and graphs.

The generic aspect of the interface API is accomplished by putting the information specific to the underlying source into the "connection". Once the connection is established, other aspects of using data are the same, so one connection can be easily interchanged with another, and so your programs do not need to be changed when the data source is changed. (But, of course, some changes will be needed if the names of the variables change.)

2.1 *TShistQuote* and *TSgetSymbol*

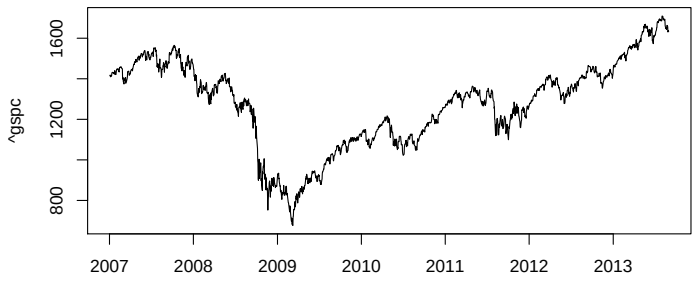
Packages *TShistQuote* and *TSgetSymbol* provide mechanisms to retrieve historical quote data from various sources. *TShistQuote* is a wrapper to *get.hist.quote* in package *tseries* (**?**). A connection to Yahoo Finance is established, and data retrieved and plotted by

```
>   library("TShistQuote")
>   yahoo <- TSconnect("histQuote", dbname="yahoo")
>   x <- TSget("^gspc", quote = "Close", con=yahoo)
>   library("tfplot")
>   tfplot(x)
```



Package *TSgetSymbol* is a wrapper to *getSymbols* in package *quantmod* (**?**). A connection to Yahoo using this, and retrieving the same data, can be done by

```
>   library("TSgetSymbol")
>   yahoo <- TSconnect("getSymbol", dbname="yahoo")
>   x <- TSget("^gspc", quote = "Close", con=yahoo)
>   tfplot(x)
```



Notice that the only difference is the library that is loaded and the name of the driver provided when establishing the connection. After that, the code is the same (and the data from the two connections should be the same). This is the coding approach one would typically follow, so that changing the data source is easy. However, sometimes it is interesting to compared the same data from differrent sources, so here is an example where the connections are given

different names, so the same data through two different connection methods can be more easily compared:

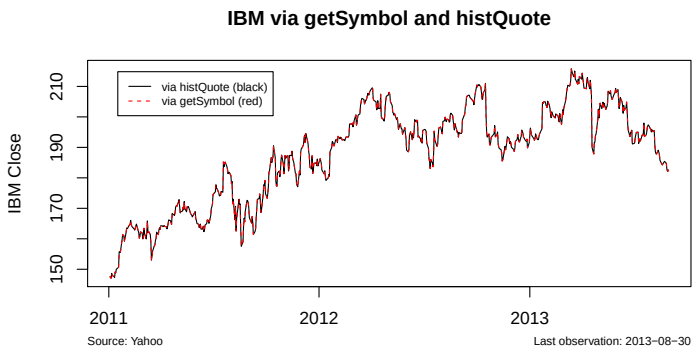
```
> ya1 <- TSconnect("getSymbol", dbname="yahoo")
> ya2 <- TSconnect("histQuote", dbname="yahoo")
> ibmC1 <- TSget("ibm", ya1, quote = "Close", start="2011-01-03")
> ibmC2 <- TSget("ibm", ya2, quote = "Close", start="2011-01-03")
```

In this example the underlying packages both return *zoo* time series objects, but the encoding of the date index vectors are of different classes (*Date* vs *POSIXct*). This is usually not a problem because one would usually work with one package or the other, but it does become a problem when comparing the two objects returned by the different methods. The time representation of *ibmC1* can be changed from *POSIXct* to *Date* by:

```
> tframe(ibmC1) <- as.Date(tframe(ibmC1))
```

The two series can then be plotted:

```
> tfplot(ibmC2, ibmC1,
     ylab="IBM Close",
     title="IBM via getSymbol and histQuote",
     lastObs=TRUE,
     legend=c("via histQuote (black)", "via getSymbol (red)"),
     source="Source: Yahoo")
```



Or the difference can be used to check equality:

```
> max(abs(ibmC2 - ibmC1))

[1] 0
```

(Note that this difference calculation does not catch a difference in length, which occurs if new data has been release on one connection and not the other. At some point Yahoo was releasing partial data early, and these connection are

correcting differently for this. So, at some times of day, the last available data point is not the same on these two connections.)

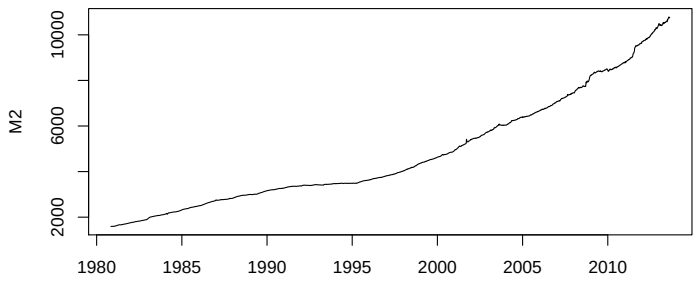
A certain amount of meta data can be returned with the time series object and can be extracted with these utilities:

```
>   TSdescription(x)

[1] "^gspc Close  from  yahoo"

>   TSdoc(x)

[1] "^gspc Close  from  yahoo retrieved  2013-09-01 17:58:24"

>   TSlabel(x)

[1] "^gspc Close"

>   TSsource(x)

[1] "yahoo"
```

2.2 *TSgetSymbol* with FRED

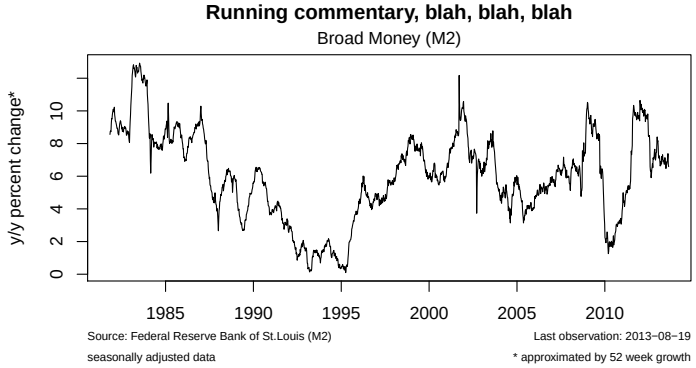
Package *TSgetSymbol* can also be used to get data from the Federal Reserve Bank of St.Louis, as will be illustrated here. (Look at `http://research.stlouisfed.org/fred/` to find series identifiers.)

```
>   library("TSgetSymbol")
>   fred <- TSconnect("getSymbol", dbname="FRED")
>   tfplot(TSget("M2", fred))
```



A connection can be specified to be used as the default, so it does not need to be specified each time:

```
>   options(TSconnection=fred)
>   tfOnePlot(percentChange(TSget("M2"), lag=52),
      title = "Running commentary, blah, blah, blah",
      subtitle="Broad Money (M2)",
      ylab= "y/y percent change*",
      source="Source: Federal Reserve Bank of St.Louis (M2)",
      footnoteLeft = "seasonally adjusted data",
      footnoteRight = "* approximated by 52 week growth",
      lastObs = TRUE )
```

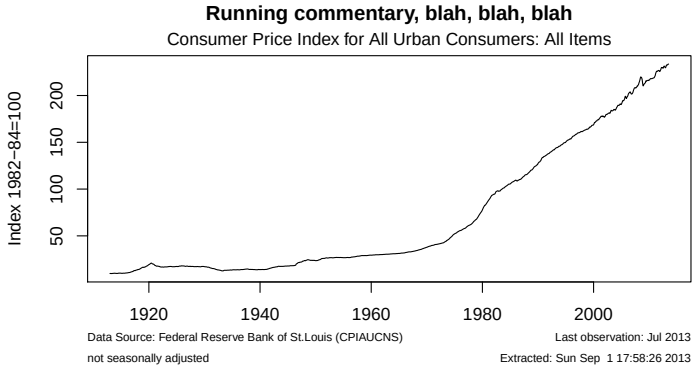


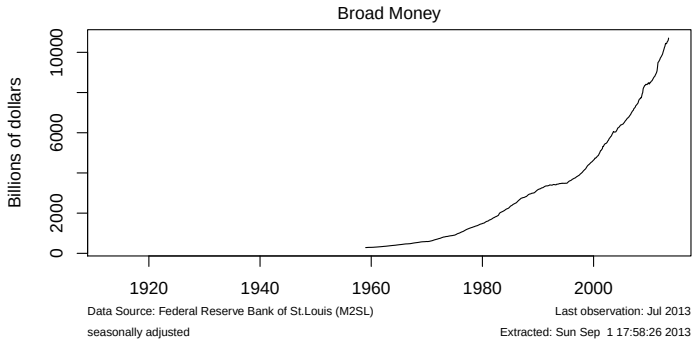
Several different connections will be used in this vignette, and so a default will not be used. To unset the default

```
>   options(TSconnection=NULL)
```

It is also possible to return multiple series, but they should all be of the same frequency. (The FRED series called M2 is a weekly series).

```
>   x <- TSget(c("CPIAUCNS","M2SL"), fred)
>   tfplot(x,
      title = "Running commentary, blah, blah, blah",
      subtitle=c("Consumer Price Index for All Urban Consumers: All Items", "Broad Money"),
      ylab= c("Index 1982-84=100", "Billions of dollars"),
      source= c("Data Source: Federal Reserve Bank of St.Louis (CPIAUCNS)",
                "Data Source: Federal Reserve Bank of St.Louis (M2SL)"),
      footnoteLeft  = c("not seasonally adjusted",  "seasonally adjusted"),
      footnoteRight = paste("Extracted:", date()),
      lastObs = TRUE )
```






```
>   TSdates(c("CPIAUCNS","M2SL"), fred)

     [,1]
[1,] "CPIAUCNS from 1913 1 to 2013 7       12"
[2,] "M2SL from 1959 1 to 2013 7       12"
```

By default, *TSget* returns *ts* time series for annual, quarterly, and monthly data, and *zoo* series otherwise. It is possible to specify the type of object to return:

```
>   x <- TSget(c("CPIAUCNS","M2SL"), fred, TSrepresentation="zoo")
>   class(x)
```

A session default can also be set for this with

```
>   options(TSrepresentation="zoo")
```

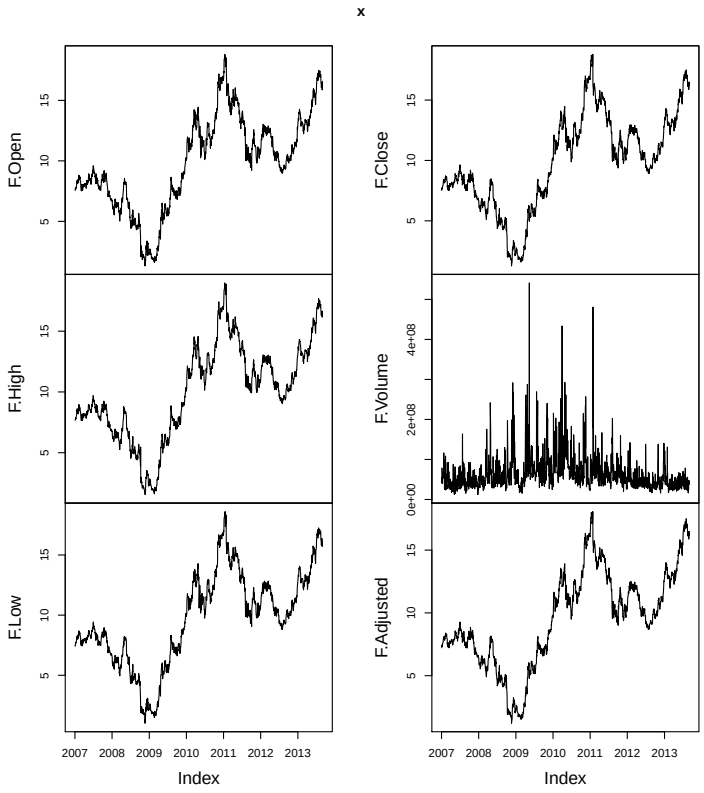
in which case all results will be return as *zoo* objects unless otherwise specified. The session default is unset with

```
>   options(TSrepresentation=NULL)
```

Beware that it does not make sense to set *ts* as the default, because it is already the default for all series that can be represented as *ts*, and will not work correctly for other series. Other representations are possible. See the *TSget* help for more details.

The following connects to yahoo and loads the ticker symbol for Ford. This is a multivariate time series with open, close, etc.

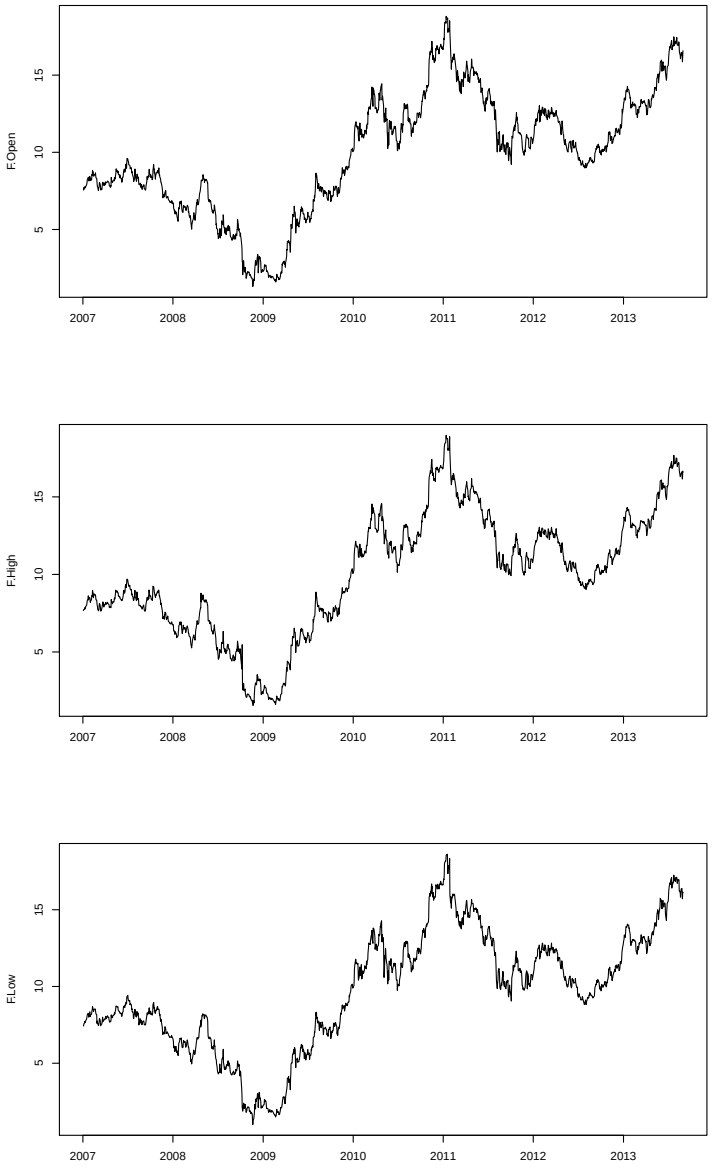
```
>   yahoo <- TSconnect("getSymbol", dbname="yahoo")
>   x <- TSget("F", con=yahoo)
>   plot(x)
```



Most of the plots in this vignette are done with the utilities in the *tfplot* package, but the usual *plot* function, used above, produces slightly different results that may be preferable in some situations. Also, for some time series objects, the plot method has been much improved from the default, so if you are using these objects you may find that *plot* provides attractive features.

In the case of the ticker data above, *tfplot* displays graphs in verticle panels. However, six panels do not nicely fit on a printed page. The first three are displayed with:

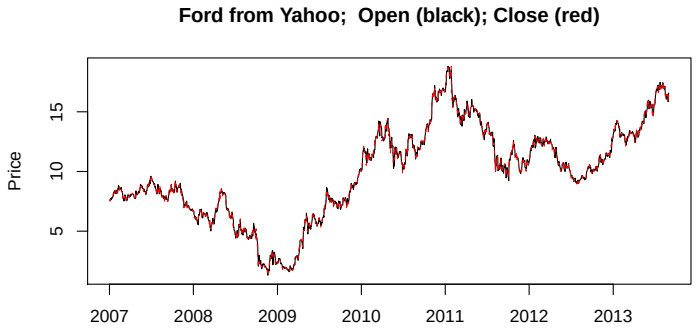
```
>   tfplot(x,series=1:3)
```



It is possible to specify the number of graphs on an ouput screen with *graphs.per.page*, for example, *tfplot(x, graphs.per.page=3)*. Set *par(ask=TRUE)* if you want to stop and prompt for <Return> between pages in the graphics output.

The *quote* argument to *TSget* can be used to specify that only a subset of the market data should be returned:

```
>   tfOnePlot(TSget("F", con=yahoo, quote=c("Open", "Close")),
       title="Ford from Yahoo;  Open (black); Close (red)",
       ylab="Price")
```



2.3 *TSjson* with Statistics Canada

Package *TSjson* provides a mechanism to extract data from websites and pass it to *R* in JavaScript Object Notation (JSON) using package *RJSONIO* (**?**). *TSjson* uses *Python* code to mechanize clicking though web pages to get a downloadable file. This really should be considered a temporary solution, until the data provider implements a true API. The current version of *TSjson* provides a connection to Statistics Canada's `http://www.statcan.gc.ca` Cansim database. (You should look at the Statistics Canada site to find series identifiers.) Please contact the package maintainer if you would like to help implement connections to other sites.

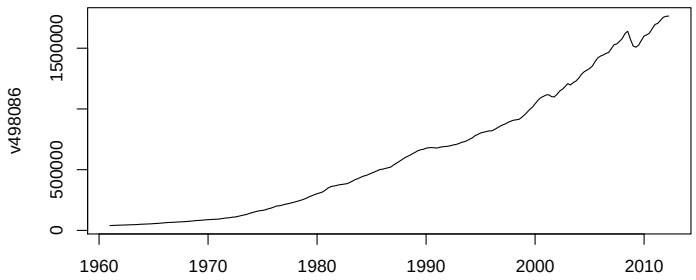
The connection can be established in two different ways. The simplest requires a system with *Python* installed, and *Python* modules *sys*, *json*, *mechanize*, *re*, *csv* and *urllib2*. Installing these may be somewhat difficult in some environments, but a second method using a proxy server can be used. The proxy server needs *Python* and the modules, as well as server software (e.g. *Web2Py*), but the client machine requires nothing special other than *R* and *TSjson*. The proxy server can be anywhere on the Internet.

The following examples use the first method. More details on establishing connections with the second method are provided in the appendix. First, establish a connection

```
> require("TSjson")
> cansim <- TSconnect("json", dbname="cansim")
```

Now data can be retrieved and a plot generated by

```
> x <- TSget("v498086", cansim)
> tfplot(x)
```

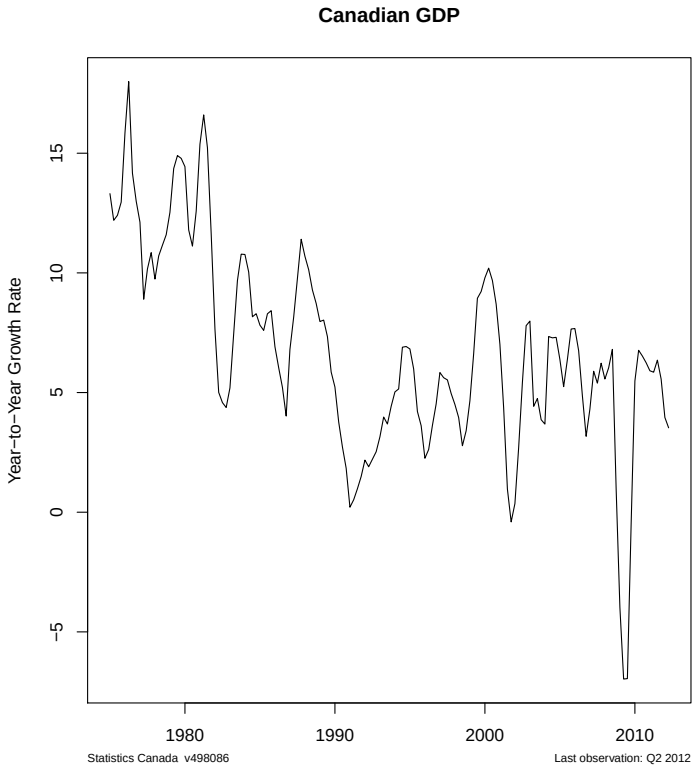


Meta data can also be retrieved:

```
> TSdescription("v498086", cansim)

[1] "Gross domestic product (GDP) at market prices (x 1,000,000)"

> TSdoc("v498086", cansim)

[1] "Table 380-0002: Gross domestic product, expenditure-based; Canada; Current prices; Seas

> TSlabel("v498086", cansim)

[1] "v498086"

> TSsource("v498086", cansim)

[1] "Statistics Canada: Table 380-0002; v498086"
```

A transformation of the data can be done, more detail added to the graph, and a start date specified:

```
> tfplot(ytoypc(x), start=c(1975,1),
     ylab="Year-to-Year Growth Rate",
     title="Canadian GDP",
     source=paste("Statistics Canada ", seriesNames(x)),
     lastObs=TRUE)
```

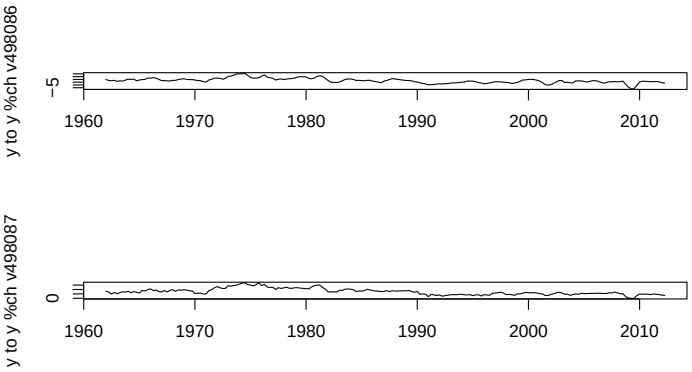


Sometimes it is useful to check availability:

```
> TSdates(c("v498086", "v498087"), cansim)

     [,1]
[1,] "v498086 from 1961 1 to 2012 2       4"
[2,] "v498087 from 1961 1 to 2012 2       4"
```

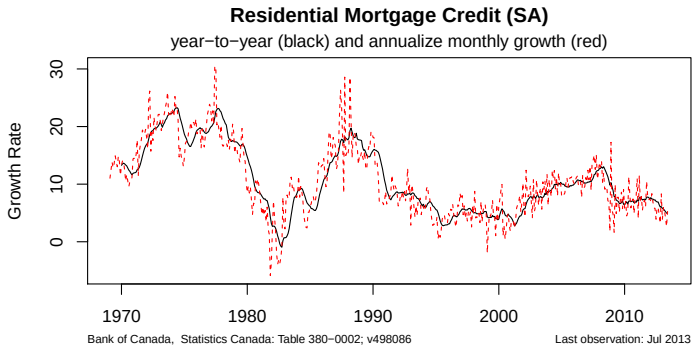
Or, plot more than one series:

```
> tfplot(ytoypc(TSget(c("v498086", "v498087"), cansim)))
```



The meta data can also be retrieved with the series, which will generally be faster than retreiving it separately, if it is needed:

```
> resMorg <- TSget("V122746", cansim, TSdescription=TRUE, TSdoc=TRUE, TSlabel=TRUE)
> TSdescription(resMorg)

[1] "Total outstanding balances (x 1,000,000)"

> TSdoc(resMorg)

[1] "Table 176-0069: Residential mortgage credit; Canada; Average at month-end; Seasonally a

> TSlabel(resMorg)

[1] "V122746"

> TSseriesIDs(resMorg)

[1] "V122746"

> TSsource(resMorg)

[1] "Statistics Canada: Table 176-0069; v122746"

> seriesNames(resMorg) <- "Residential Mortgage Credit (SA)"
> tfplot(ytoypc(resMorg), annualizedGrowth(resMorg),
     title=seriesNames(resMorg),
     subtitle="year-to-year (black) and annualize monthly growth (red)",
     ylab="Growth Rate",
     source=paste("Bank of Canada, ", TSsource(x)),
     lastObs=TRUE)
```



2.4 TSxls

TSxls provides methods for the *TSdbi* interface, allowing the use of spreadsheets as if they are a database. (This is a poor substitute for a real database, but is sometimes convenient.) *TSxls* uses package *read.xls* in *gdata* (**?**) *TSxls* does not support writing data to the spreadsheet (but to write time series data to a spreadsheet see *TSwriteXLS* in *tframePlus*, discussed in section 7). The spreadsheet can be a remote file, which is retrieved when the connection is established.

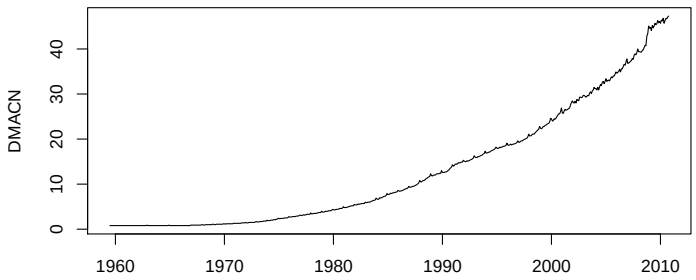
The following retrieves a file from the Reserve Bank of Australia and maps the elements that are used: data, dates, identifiers, and series names.

```
>   library("TSxls")
>   rba <- TSconnect("xls",
          dbname="http://www.rba.gov.au/statistics/tables/xls/d03hist.xls",
          map=list(ids  =list(i=11,     j="B:Q"),
                   data =list(i=12:627, j="B:Q"),
                   dates=list(i=12:627, j="A"),
                   names=list(i=4:7,    j="B:Q"),
                   description = NULL,
                   tsrepresentation = function(data,dates){
                       ts(data,start=c(1959,7), frequency=12)}))
```

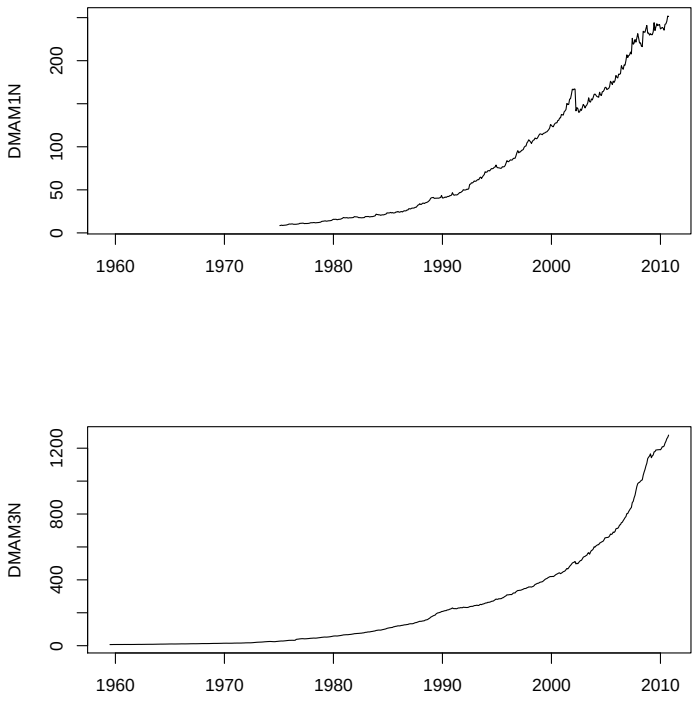
This also illustrates how *tsrepresentation* can be specified as an arbitrary function to set the returned time series object representation.

Beware that data is read into R when the connection is established, so changes in the spreadsheet will not be visible in R until a new connection is established.

```
>   x <- TSget("DMACN", rba)
>   require("tfplot")
>   tfplot(x)
```



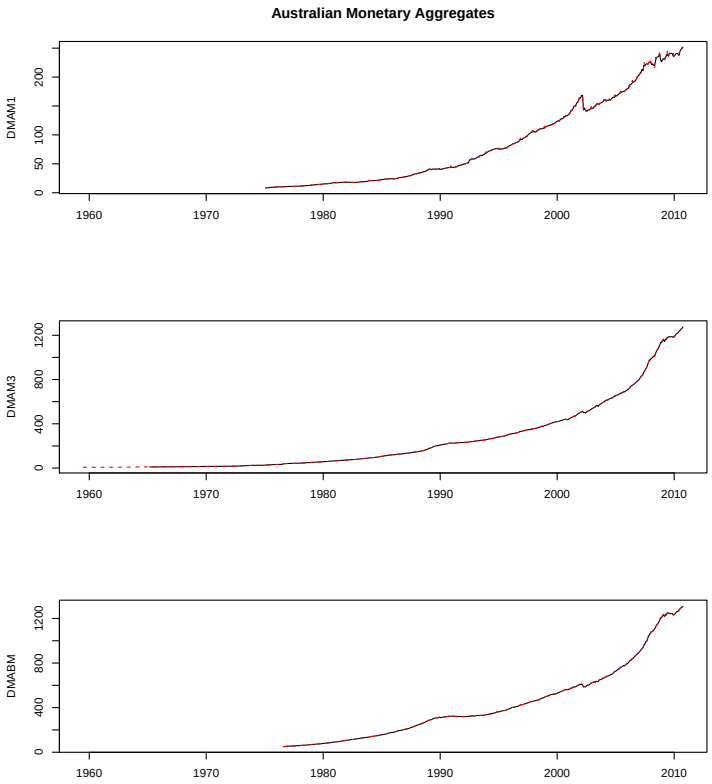

```
>   x <- TSget(c("DMAM1N", "DMAM3N"), rba)
>   tfplot(x)
>   TSdescription(x)

[1] "  from  http://www.rba.gov.au/statistics/tables/xls/d03hist.xls"
[2] "  from  http://www.rba.gov.au/statistics/tables/xls/d03hist.xls"
```



tfplot treats each series in the first argument as a panel to be plotted. It is possible to specify the number of graphs on each page of the output device with the argument *graphs.per.page*. As previously illustrated, it is also possible to specify that a subset of the series should be selected. (Also, as already illustrated above, the function *plot* displays the series somewhat differently than *tfplot*, and possibly differently depending on the objects time series representation.)

tfplot takes additional time series objects as arguments. Series in the first argument are plotted in separate panels. Series in subsequent time series objects will be plotted respectively on the same panels as the first, so the number of series in each object must be the same.

```
>   tfplot(TSget(c("DMAM1S", "DMAM3S", "DMABMS"), rba),
           TSget(c("DMAM1N", "DMAM3N", "DMABMN"), rba),
      ylab=c("DMAM1", "DMAM3", "DMABM"),
      title="Australian Monetary Aggregates")
```

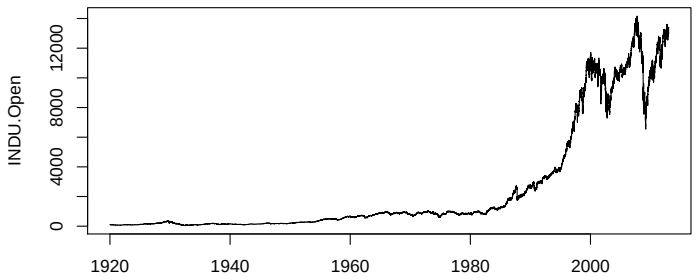


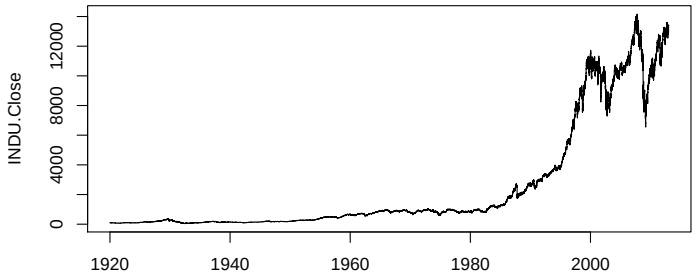
2.5 TSzip

TSzip provides methods for the *TSdbi* interface, allowing the use of zipped files that can be read by *read.table* as if each file is a database series (or group of series such as high, low, open, close, for a stock). The *dbname* is a directory or *url*. *TSzip* does not support writing data to the database.

The following retrieves zipped files from `http://pitrading.com/free_market_data.htm` which provides some end of day data free of charge. (Disclaimer: This site is used as an example. Other than using this free data, I have no association with the company.)

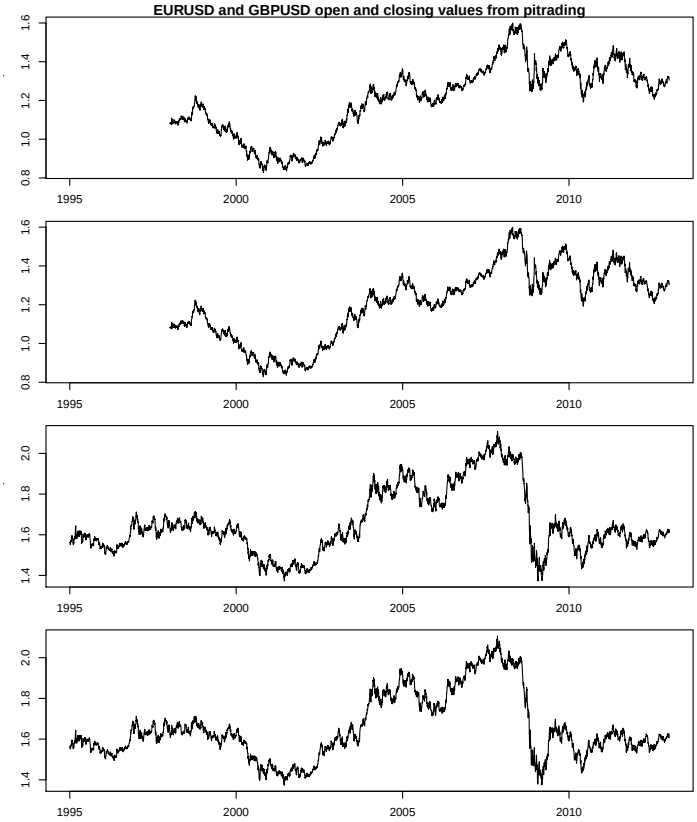
```
>   library("TSzip")
>   pitr <- TSconnect("zip", dbname="http://pitrading.com/free_eod_data")
>   z <- TSget("INDU", pitr)
>   tfplot(z, series=c(1,4))
```





The following illustrates returning an *xts* (**?**) time series object.

```
>   z <- TSget(c("EURUSD", "GBPUSD"), pitr, quote=c("Open","Close"),
          TSrepresentation=xts)
>   tfplot(z,
          title="EURUSD and GBPUSD open and closing values from pitrading",
          start="1995-01-01",
          par=list(omi=c(0.1,0.3,0.1,0.1),mar=c(2.1,3.1,1.0,0.1)))
```



The default appearance of graphs can be changed (improved) by adjusting graphics device margins *omi* and *mar*. (They are set by the vector in order: bottom, left, top, right.) The default behaviour is a compromise that usually works reasonally well for both screen and printed output. It is often useful to adjust these when generating *pdf* files for publication.

3 SQL Time Series Databases

This section gives several simple examples of putting series on and reading them from a database. (If a large number of series are to be loaded into a database, one would typically do this with a batch process using the database program's utilities for loading data.) The examples in this section will use *TSMySQL* but, other than the initial connection, access will be similar for other SQL *TS** packages. The syntax for connecting with other packages, and other options for connecting with *TSMySQL*, are provided in Appendix "A".

The packages *TSPostgreSQL*, *TSMySQL*, *TSSQLite*, and *TSodbc* use underlying packages *RPostgreSQL* (**?**), *RMySQL* (**?**), *RSQLite* (**?**), and *RODBC* (**?**). The *TS** packages provide access to an SQL database with an underlying table structure that is set up to store time series data.

The next lines of code do some preliminary setup of the database. This uses the underlying database connection (*dbConnect*) rather than *TSconnect*, because *TSconnect* will not recognize the database until it has been setup.

WARNING: running this will overwrite the "test" database on your server.

```
> library("TSMySQL")
> con <- dbConnect("MySQL", dbname="test")
> source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
> dbDisconnect(con)
```

3.1 Writing to SQL Databases

This subsection illustrates writing some simple artifical data to a database, and reading it back. This part of the vignette is generated using *TSMySQL*, but other backend SQL servers work in a similar way. See Appendix "A" for details of establishing other SQL database connections.

The first thing to do is to establish a *TSdbi* connection to the database:

```
> library("TSMySQL")
> con <- TSconnect("MySQL", dbname="test")
```

TSconnect uses *dbConnect* from the *DBI* package, but checks that the database has expected tables, and checks for additional features. (It cannot be used before the tables are created, as was done above.)

The follow illustrates the use of the *TSdbi* interface, which is common to all extension packages.

This puts a series called *vec* on the database and then reads is back

```
> z <- ts(rnorm(10), start=c(1990,1), frequency=1)
> seriesNames(z) <- "vec"
> if(TSexists("vec", con)) TSdelete("vec", con)
> TSput( z, con)
> z <- TSget("vec", con)
```

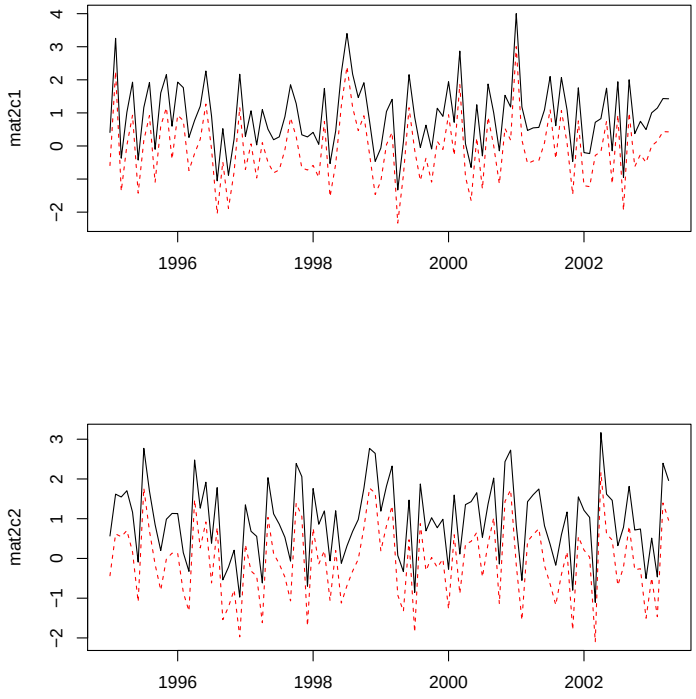
Note that the series name(s) and not the *R* variable name (in this case, *vec* not *z*) are used on the database. If the retrieved series is printed it is seen to be a "ts" time series with some extra attributes.

TSput fails if the series already exists on the *con*, so the above example checks and deletes the series if it already exists. *TSreplace* does not fail if the series does not yet exist, so examples below use it instead. *TSput*, *TSdelete*, *TSreplace*, and *TSexists* all return logical values *TRUE* or *FALSE*.

Several plots below show original data and the data retrieved after it is written to the database. In the plot below, one is added to the original data so that both lines are visible.

The *R* variable can contain multiple series of the same frequency. They are stored separately on the database.

```
> z <- ts(matrix(rnorm(200),100,2), start=c(1995,1), frequency=12)
>  seriesNames(z) <- c("mat2c1", "mat2c2")
> TSreplace(z, con)

[1] TRUE

> tfplot(z+1, TSget(c("mat2c1","mat2c2"), con),
            lty=c("solid", "dashed"), col=c("black", "red"))
```



The following extract information about the series from the database, although not much information has been added for these examples.

```
> TSmeta("mat2c1", con)

serIDs:  mat2c1
 from dbname  test using  TSMySQLConnection

> TSmeta("vec", con)

serIDs:  vec
 from dbname  test using  TSMySQLConnection
```

```
> TSdates("vec", con)

     [,1]
[1,] "vec from 1990 1 to 1999 1 A      "

> TSdescription("vec", con)

[1] NA

> TSdoc("vec", con)

[1] NA

> TSlabel("vec", con)

[1] NA
```

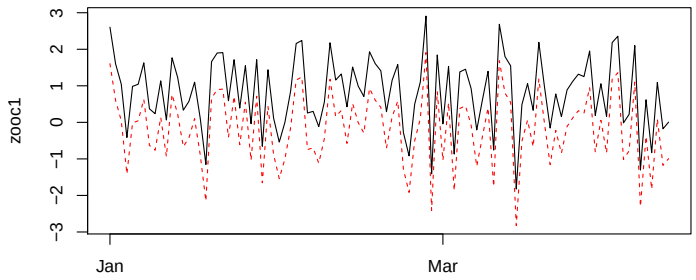
Data documentation can be in three forms. A description specified by *TSdescription*, longer documentation specified by *TSdoc*, or a short label, typically useful on a graph, specified by *TSlabel*. These can be added to the time series object, in which case they will be written to the database when *TSput* or *TSreplace* is used to put the series on the database. Alternatively, they can be specified as arguments to *TSput* or *TSreplace*. The description or documentation will be retrieved as part of the series object with *TSget* only if this is specified with the logical arguments *TSdescription* and *TSdoc*. They can also be retrieved directly from the database with the functions *TSdescription* and *TSdoc*.

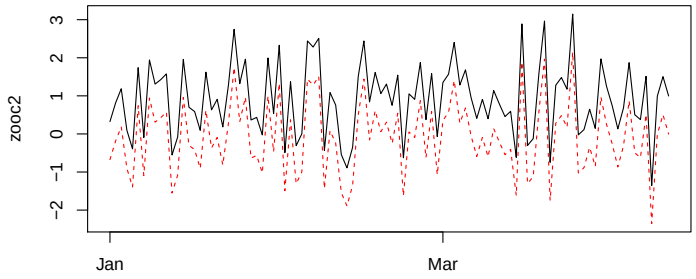
```
> z <- ts(matrix(rnorm(10),10,1), start=c(1990,1), frequency=1)
> TSreplace(z, serIDs="Series1", con)

[1] TRUE

> zz <- TSget("Series1", con)
> TSreplace(z, serIDs="Series1", con,
      TSdescription="short rnorm series",
      TSdoc="Series created as an example in the vignette.")

[1] TRUE

> zz <- TSget("Series1", con, TSdescription=TRUE, TSdoc=TRUE)
> start(zz)

[1] 1990    1

> end(zz)

[1] 1999    1

> TSdescription(zz)
```

```
[1] "short rnorm series"

> TSdoc(zz)

[1] "Series created as an example in the vignette."

> TSdescription("Series1", con)

[1] "short rnorm series"

> TSdoc("Series1", con)

[1] "Series created as an example in the vignette."
```

The following examples use dates and times which are not handled by *ts*, so the *zoo* time representation is used. It is necessary to specify the table where the data should be stored in cases where it is difficult to determine the periodicity of the data. See Appendix "B" for details of the specific tables.

```
> require("zoo")
> z <- zoo(matrix(rnorm(200),100,2), as.Date("1990-01-01") + 0:99)
>  seriesNames(z) <- c("zooc1", "zooc2")
> TSreplace(z, con, Table="D")

[1] TRUE

> tfplot(z+1, TSget(c("zooc1","zooc2"), con),
           lty=c("solid", "dashed"), col=c("black", "red"))
>
```






```
>  z <- zoo(matrix(rnorm(200),100,2), as.Date("1990-01-01") + 0:99 * 7)
>   seriesNames(z) <- c("zooWc1", "zooWc2")
>  TSreplace(z, con, Table="W")

[1] TRUE

>   dbDisconnect(con)
>   detach(package:TSMySQL)
>   detach(package:RMySQL)
```

4 More Examples

4.1 Examples Using SQL Databases

This section illustrates fetching data from the web and loading it into the database. This would be a very slow way to load a database, but provides examples of different kinds of time series data. The fetching is done with *TShistQuote*. Fetching data can fail due to lack of an Internet connection or delays, which will cause the generation of this vignette to fail.

This part of the vignette is generated using *TSPostgreSQL*, but other backend SQL servers work in a similar way. See Appendix "A" for details of establishing other SQL database connections.

First establish a connection to the database where data will be saved:

```
>   require("TSPostgreSQL")
>   host    <- Sys.getenv("POSTGRES_HOST")
>   con <-  TSconnect("PostgreSQL", dbname="test", host=host)
```

Now connect to the web server and fetch data:

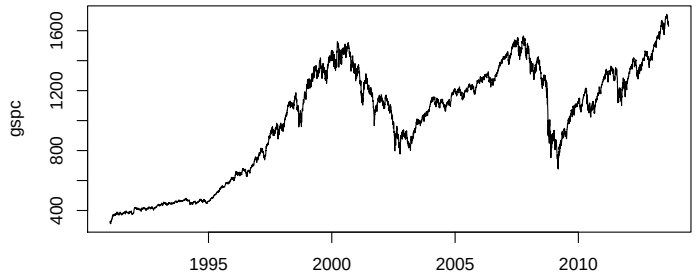
```
>   require("TShistQuote")
>   yahoo <- TSconnect("histQuote", dbname="yahoo")
>   x <- TSget("^gspc", quote = "Close", con=yahoo)
```

Then write the data to the local server, specifying table B for business day data (using TSreplace in case the series is already there from running this example previously):

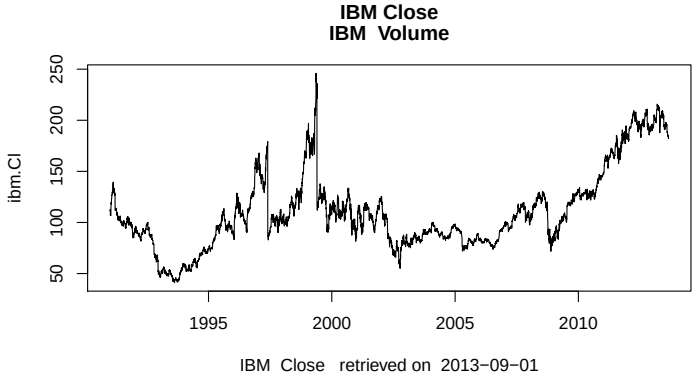
```
>    TSreplace(x,  serIDs="gspc", Table="B", con=con)

[1] TRUE
```

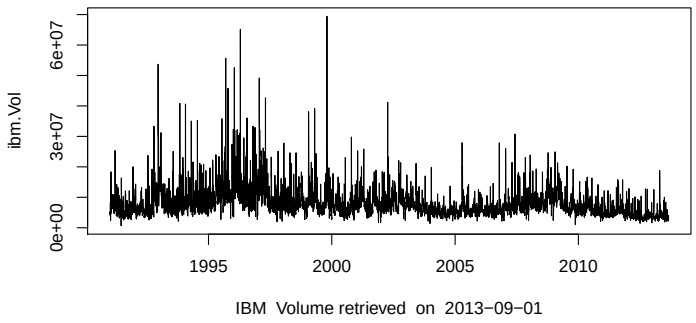
and check the saved version:

```
>   TSrefperiod(TSget(serIDs="gspc", con=con))

[1] "Close"

>   TSdescription("gspc", con=con)

[1] "^gspc Close  from  yahoo"

>   TSdoc("gspc", con=con)

[1] "^gspc Close  from  yahoo retrieved  2013-09-01 18:09:54"

>   tfplot(TSget(serIDs="gspc", con=con))
```



```
>    x <- TSget("ibm", quote = c("Close", "Vol"), con=yahoo)
>    TSreplace(x, serIDs=c("ibm.Cl", "ibm.Vol"),  con=con, Table="B",
               TSdescription.=c("IBM Close","IBM  Volume"),
               TSdoc.= paste(c(
                  "IBM  Close   retrieved on ",
                  "IBM  Volume retrieved  on "), Sys.Date()))
[1] TRUE
>    z <- TSget(serIDs=c("ibm.Cl", "ibm.Vol"),
              TSdescription=TRUE, TSdoc=TRUE, con=con)
>    TSdescription(z)
[1] "IBM Close"   "IBM  Volume"
>    TSdoc(z)
[1] "IBM  Close   retrieved on  2013-09-01"
[2] "IBM  Volume retrieved  on  2013-09-01"
>    tfplot(z, xlab = TSdoc(z), title = TSdescription(z))
>    tfplot(z, title="IBM", start="2007-01-01")
```





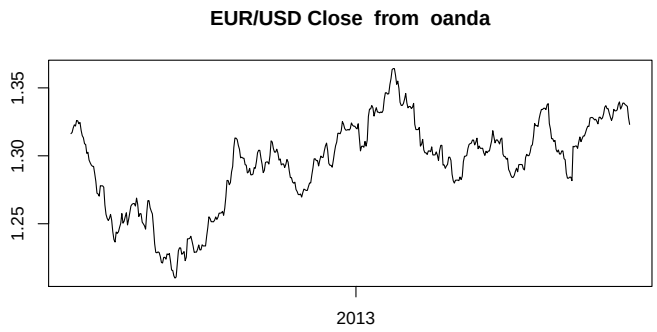
Oanda has maximum of 500 days, so the start date is specified here so as to not exceed that.

```
>    Oanda <- TSconnect("histQuote", dbname="oanda")
>    x <- TSget("EUR/USD", start=Sys.Date() - 495, con=Oanda)
>    TSreplace(x, serIDs="EUR/USD", Table="D", con=con)
```

```
[1] TRUE
```

Then check the saved version:

```
>    z <- TSget(serIDs="EUR/USD",TSlabel=TRUE,
                    TSdescription=TRUE,  con=con)
>    tfplot(z, title = TSdescription(z), ylab=TSlabel(z),
             start="2007-03-01")
```




```
>    dbDisconnect(con)
>    dbDisconnect(yahoo)
>    dbDisconnect(Oanda)
>    detach(package:TSPostgreSQL)
>    detach(package:RPostgreSQL)
```

5 Comparing Time Series Databases

The purpose of package *TScompare* is to compare pairs of series on two database. These series might have the same name, but for generality the main function, *TScompare*, is set up to use name pairs. The pairs to compare are indicated by a matrix of strings with two columns. (It would also be possible to compare pairs on the same database, which might make sense if the names are different.)

The connections are established using other *TSdbi* packages such as *TSMySQL*, *TSPostgreSQL*, etc. It will be necessary to establish two database connections, so it will also be necessary to load the database specific packages. Examples in this vignette use *TShistQuote*, *TSMySQL* and *TSSQLite*.

```
> library("TScompare")
> library("TShistQuote")
> library("TSMySQL")
> library("TSSQLite")
```

First setup database tables that are used by TSdbi using a *dbConnect* connection, after which a *TSconnect* connection can be used (This is re-done to insure the database are empty):

```
>  con <- dbConnect("MySQL", dbname="test")
>  source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
>  dbDisconnect(con)
>  con <- dbConnect("SQLite", dbname="test")
>  source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
>  dbDisconnect(con)
```

Now TS connections to the databases are established.

```
>   con1 <- TSconnect("MySQL", dbname="test")
>   con2 <- TSconnect("SQLite", dbname="test")
```

Next a connection to yahoo is used to get some series and write them to the local test database. *TSreplace* is used because *TSput* will fail if the series already exisits.

```
>   yahoo <- TSconnect("histQuote", dbname="yahoo")
>   x <- TSget("^ftse", yahoo)
>   TSreplace(x, serIDs="ftse", Table="B", con=con1)

[1] TRUE

>   TSreplace(x, serIDs="ftse", Table="B", con=con2)

[1] TRUE

>   x <- TSget("^gspc", yahoo)
>   TSreplace(x,  serIDs="gspc", Table="B", con=con1)

[1] TRUE

>   TSreplace(x,  serIDs="gspc", Table="B", con=con2)

[1] TRUE

>   x <- TSget("ibm", con=yahoo, quote = c("Close", "Vol"))
>   TSreplace(x, serIDs=c("ibmClose", "ibmVol"), Table="B", con=con1)

[1] TRUE

>   TSreplace(x, serIDs=c("ibmC",     "ibmV"),   Table="B", con=con2)
```

```
[1] TRUE
```

Now to do a comparison:

```
>   ids <- AllIds(con1)
>   ids

[1] "ftse"     "gspc"     "ibmClose" "ibmVol"
```

If the second database has the same names then ids can be made into a matrix with identical columns.

```
>   ids <- cbind(ids, ids)
>   eq   <- TScompare(ids, con1, con2, na.rm=FALSE)
>   summary(eq)

4  of  4 are available on con1.
2  of  4 are available on con2.
2  of  2 remaining have the same window.
2  of  2 remaining have the same window and values.

>   eqrm <- TScompare(ids, con1, con2, na.rm=TRUE)
>   summary(eqrm)

4  of  4 are available on con1.
2  of  4 are available on con2.
2  of  2 remaining have the same window.
2  of  2 remaining have the same window and values.
```

Since names are not identical the above indicates discrepancies, which are resolves by indicating the corresponding name pairs:

```
>   ids <- matrix(c("ftse","gspc","ibmClose", "ibmVol",
                    "ftse","gspc","ibmC", "ibmV"),4,2)
>   ids

     [,1]       [,2]
[1,] "ftse"     "ftse"
[2,] "gspc"     "gspc"
[3,] "ibmClose" "ibmC"
[4,] "ibmVol"   "ibmV"

>   eq   <- TScompare(ids, con1, con2, na.rm=FALSE)
>   summary(eq)

4  of  4 are available on con1.
4  of  4 are available on con2.
4  of  4 remaining have the same window.
4  of  4 remaining have the same window and values.
```

```
>   eqrm <- TScompare(ids, con1, con2, na.rm=TRUE)
>   summary(eqrm)

4  of  4 are available on con1.
4  of  4 are available on con2.
4  of  4 remaining have the same window.
4  of  4 remaining have the same window and values.
```

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>   dbDisconnect(con1)
>   dbDisconnect(con2)
>   dbDisconnect(yahoo)
>   detach(package:TSMySQL)
>   detach(package:RMySQL)
>   detach(package:TSSQLite)
>   detach(package:RSQLite)
```

6 Vintages of Realtime Data

Examples in the section have been disabled pending availability of different dataset.

Data vintages, or "realtime data" are snapshots of data that was available at different points in time. The most obvious feature of earlier snapshots is that the series end earlier. However, the reason for retaining vintages is that data is often revised, so, for some observations, earlier vintages have different data. Typically most revisions happen for the most recent periods, but this is often the data of most interest for forcasting and policy decisions. Thus, the revision records are valuable for understanding the implications of differences between early and revised releases of the data. A simple mechanism for accessing vintages of data is available in several *TS** packages. This is illustrated here with an SQL database that has been set up with vintage support.

First establish a connection to th database and get a vector of the available vintages:

```
> require("TSMySQL")
> require("tfplot")
> ets <- TSconnect("MySQL",dbname="etsv")
> v  <- TSvintages(ets)
```

The following checks if a certain variable (Consumer Credit – V122707) is available in the different vintages. (V numbers replaced B numbers circa 2003, so the V numbers do not exist in older vintages. This could be supported by implementing aliases, but that has not been done here.)

```
> ve <- TSexists("V122707", vintage=v, con=ets)
```

```
> ve[224:length(ve)] <- FALSE
```

The vintages can them be retrieved and plotted by

```
> CC <- TSget(serIDs="V122707", con=ets, vintage=v[ve])
> tfOnePlot(ytoypc(CC), start=c(2000,1),
     ylab="Consumer Credit (V122707) y/y Growth",
     title=paste("Vintages", v[ve][1], "to",  v[ve][189]),
     lastObs=TRUE, source="Source: Bank of Canada")
```

With package *TSfame* vintages are supported if the vintages are stored in files with names like "etsmfacansim_20110513.db". Then the vintages can be accessed as follows:

```
>   dbs <- paste("ets /path/to/etsmfacansim_", c(
              "20110513.db", "20060526.db", "20110520.db"), sep="")
>   names(dbs) <- c("2011-05-13", "2006-05-26", "2011-05-20")
>   conetsV <-TSconnect("fame", dbname=dbs, "read", current="2011-05-13")
>   z <- TSget("V122646", con=conetsV, vintage=c("2011-05-13", "2006-05-26"))
>   dbDisconnect(conetsV)
```

(The above example should work, but beware that I am no longer testing it because I no longer have Fame access.)

The package *googleVis* can be used to produce a plot that is very useful for examining vintage more closely, and finding outliers and other data problems. The names are used in the legend of this next plot, so the series names are specified in the argument to *ytoypc*. (Otherwise they get reset to indicate the year-to-year calculation, which makes the legend messy to read.)

```
> require("googleVis")
> tfVisPlot(ytoypc(CC, names=seriesNames(CC)), start=c(2006,1),
      options=list(title="Vintages of Consumer Credit (V122707) y/y Growth"))
```

This will produce a graph in your web browser. It is not reproduced here. (And beware that it may not be very fast.) Pointing your mouse at the legend of this plot will highlight the corresponding vintage, and pointing at the graph will give information about the source of a data point.

```
>   dbDisconnect(ets)
>   detach(package:TSMySQL)
>   detach(package:RMySQL)
```

7 Just Want Data in a xls or csv File

Occasionally one may need to get data into a program other than *R*. Or, perhaps you have a friend that does not want to use *R* but would like easy access to data. This section describes two utilities for putting time series data obtained with

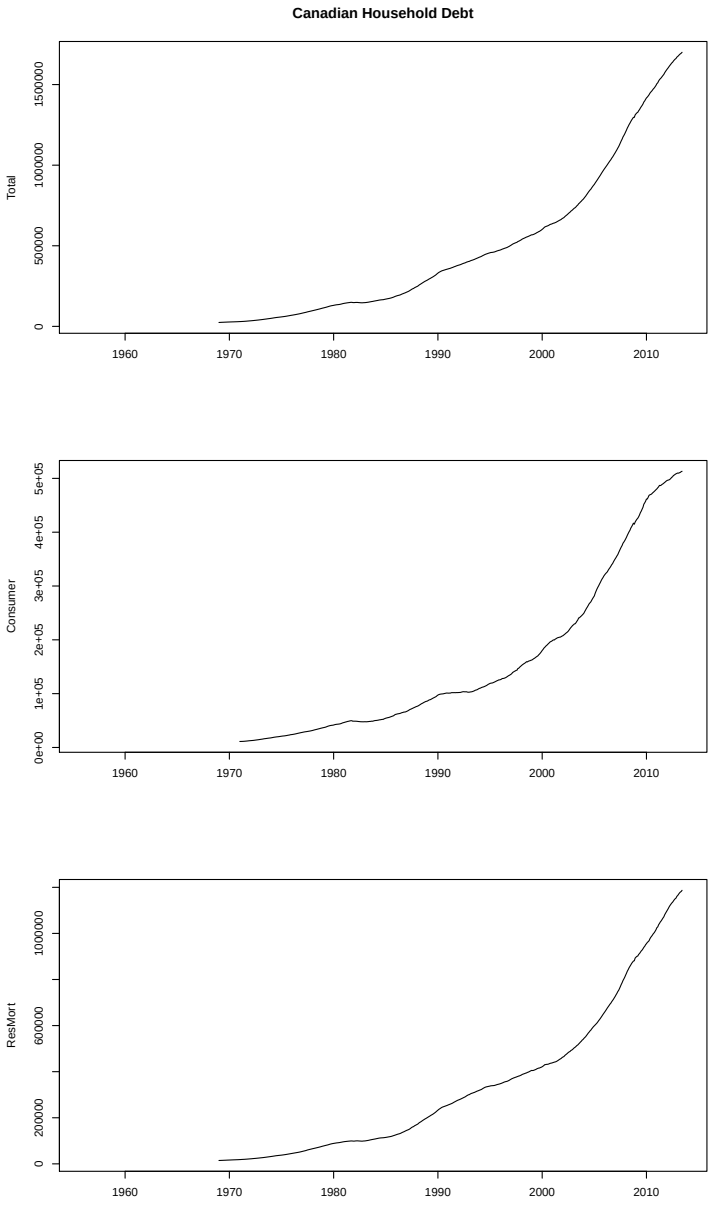
TSdbi connections into .xls and .csv files. Since it is often straight forward to downlaod an xls or csv file from a web site in the first place, the advantage of this will usually be only in the case where you need to repeatedly get the data. In that case, the process can be automated with these tools.

The starting point will be to get some data. Previous sections illustrate several possibilities. For example purposes here use

```
> require("TSjson")
> cansim <- TSconnect("json", dbname="cansim")
> z <- TSget(c("V36415","V122707","V122746"),  con=cansim)
> seriesNames(z) <- c("Total", "Consumer", "ResMort")
```

While this is optional, it might be useful to look at the data to see if it is really what you expect:

```
> library("tframePlus")
> library("tfplot")
> tfplot(z, title="Canadian Household Debt")
```



In this case the dates of one series start in 1956, but data is NA until about 1970. It is possible to trim the *NA* data from the beginning with

```
> z <- trimNA(z, endNAs=FALSE)
```

(*trimNA* will trim all series if any one of them is *NA*, so trimming the end will not be what you want if you are interested in the most recent data.)

It is also possible to trim the data to a specified subsample with

```
> z <- tfwindow(z, start=c(1995,1))
```

Now write the data to an .xls file using *TSwriteXLS* which uses *WriteXLS* (**?**) but automatically adds time information:

```
> library("WriteXLS")
> tofile <- tempfile(fileext = ".xls")
> TSwriteXLS(z, FileName=tofile)
> unlink(tofile)
```

In this case a temporary file is written, then removed with *unlink*, so that scratch files do not accumulate from building this vignette, but you would typically use a more meaningful name, and not remove the file. It is also possible to write multiple sheets in the file. For more details on this see the help

```
> ?TSwriteXLS
```

The dates are written in some different formats in the .xls file, since this is often convenient for calculations and graphing in a spreadsheet program. Beware that .xls files have size limitations that you might encounter if you try to put large amounts of data into the file. Consider using .csv files in that case.

To write the data to a .csv file

```
> tofile <- tempfile(fileext = ".csv")
> TSwriteCSV(z, FileName=tofile)
> unlink(tofile)
```

8 Appendix A: Connection Specific Details

This section provides details of the different connections which are specific to individual packages and backend databases. In order to make the examples complete, for the SQL versions, test databases are first created with the tables expected by the *TS** packages. Note that this is done with a *dbConnect* connection rather than a *TSconnect* connection, because *TSconnect* expects the tables to exist already.

WARNING: running these example will overwrite tables in the "test" database on the server.

The database setup might typically be done by an administrator, rather than by an end user. Here it is done using a small script *CreateTables.TSsql* distributed with the *TSdbi* package. A more detailed description of the instructions for building the database tables is given in the vignette for the *TSdbi* package. Those instruction show how to build the database using database utilites rather than *R*, which might be the way a system administrator would build the database.

In many cases there are two or more ways to pass information like the *username*, *password*, and *server* or *host*. One mechanism is that this information is specified in a configuration file in the user's home directory. The database driver then reads this information and it is not part of the user's *R* session. (Often this is considered the most secure way.) Another way is that environment variables are set, and the database driver uses these. Again, this is not part of the user's R session. Still another way is that the user passes this information in the call to *TSconnect* in their *R* session. In this case the character strings are visible in the R session, and possibly recorded in the user's R scripts, thus this is typically not considered to be very secure. A modification, which is only a little bit better, is for the user's *R* scripts to read the information from environment variables using, for example:

```
>    user <- Sys.getenv("MYSQL_USER")
```

8.1 TSMySQL Connection Details

The MySQL user, password, and hostname should be set in MySQL client configuration file (.my.cnf) in the user's home directory before starting R. Alternatively, this information can be set with environment variables MYSQL_USER, MYSQL_PASSWD and MYSQL_HOST. (An environment variable MYSQL_DATABASE can also be set, but "test" is specified below.) Below the configuration file is used.

The next small section of code uses *dbConnect* to set up database tables that expected by *TSconnect*.

```
>  library("TSMySQL")
>  con <-  dbConnect("MySQL", dbname="test")
>  source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
>  dbDisconnect(con)
```

Now a TSdbi connection to the database is established.

```
>   con  <-  TSconnect("MySQL", dbname="test")
```

The alternative to pass the user/password information in the arguments to the connection function would be:

```
> con <-  TSconnect("MySQL", dbname="test", username=user, password=passwd, host=host)
```

This is may be cumbersome to change, and is generally considered to be less secure.

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>   detach(package:TSMySQL)
>   detach(package:RMySQL)
```

8.2 TSPostgreSQL Connection Details

The PostgreSQL user, and password, can be set in PostgreSQL configuration file (.pgpass in Linux) in the user's home directory before starting R. The Postgress documentation suggests that it should be possible to get the host from the .pgpass file too, but I have not been able to make that work. The PostgreSQL alternative to the configuration file is to use environment variables PGDATABASE, PGHOST, PGPORT, and PGUSER. This package supports another alternatively to set this information with environment variables POSTGRES_USER, POSTGRES_PASSWD and POSTGRES_HOST, which are read in the R code. (An environment variable POSTGRES_DATABASE can also be set, but "test" is specified below.) Below, the environment variable POSTGRES_HOST is used to determine the host server, but the .pgpass file is used for the user and password information.

```
>    host    <- Sys.getenv("POSTGRES_HOST")
```

The next small section of code uses *dbConnect* to set up database tables that expected by *TSconnect*.

```
>   library("TSPostgreSQL")
>   con  <- dbConnect("PostgreSQL", dbname="test", host=host)
>   source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
>   dbDisconnect(con)
```

Now a TSdbi connection to the database is established.

```
>   con  <-  TSconnect("PostgreSQL", dbname="test", host=host)
```

Another alternative is to pass the user/password information in the arguments to the connection function:

```
>   con <-  TSconnect("PostgreSQL", dbname="test", user=user, password=passwd, host=host)
```

This is may be cumbersome to change, and is generally considered to be less secure.

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>   detach(package:TSPostgreSQL)
>   detach(package:RPostgreSQL)
```

8.3 TSSQLite Connection Details

In SQLite there does not seem to be any need to set user or password information, and examples here all use the localhost.

Now setup database tables that are used by TSdbi using a *dbConnect* connection, after which a *TSconnect* connection can be used:

```
> library("TSSQLite")
> con <- dbConnect("SQLite", dbname="test")
> source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
> dbDisconnect(con)
```

Now a TSdbi connection to the database is established.

```
>  con  <-  TSconnect("SQLite", dbname="test")
```

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>  detach(package:TSSQLite)
>  detach(package:RSQLite)
```

8.4 TSodbc Connection Details

The ODBC user, password, hostname, etc, should be set in ODBC client configuration file in the user's home directory (.odbc.ini in Linux) before starting R. An example of this file is provided below. It will also be necessary to have the appropriate driver installed on the system (Postgresql in the example below). Alternatively, it should be possible to set this information with environment variables ODBC_USER, ODBC_PASSWD and ODBC_DATABASE. However, the variable ODBC_HOST does not seem to work for passing the ODBC connection, so a properly setup ODBC configuration file is needed. Because of this, the environment variable mechanism is not currently supported in *TSodbc* and the user, passwd, and host settings should be done in the configuration file.

Now setup database tables that are used by TSdbi using a *odbcConnect* connection, after which a *TSconnect* connection can be used:

```
> library("TSodbc")
> con <-  odbcConnect(dsn="test")
> if(con == -1) stop("error establishing ODBC connection.")
> source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
> odbcClose(channel=con)
```

Now a TSdbi connection to the database is established.

```
>  con  <-  TSconnect("ODBC", dbname="test")
```

Another alternative is to pass the user/password information in the arguments to the connection function:

```
> con  <-  TSconnect("ODBC", dbname="test", uid=user, pwd=passwd)
```

This is may be cumbersome to change, and is generally considered to be less secure.

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>  detach(package:TSodbc)
>  detach(package:RODBC)
```

8.4.1 Example ODBC configuration file

Following is an example ODBC configuration file I use in Linux (so the file is in my home directory and called ".odbc.ini") to connect to a remote PostgreSQL server:

```
[test]

Description             = test DB (Postgresql)
Driver                  = Postgresql
Trace                   = No
TraceFile               = /tmp/test_odbc.log
Database                = test
Servername              = some.host
UserName                = paul
Password                = mySecret
Port                    = 5432
Protocol                = 6.4
ReadOnly                = No
RowVersioning           = No
ShowSystemTables        = No
ShowOidColumn           = No
FakeOidIndex            = No
ConnSettings            =


[ets]

Description             = ets DB (Postgresql)
Driver                  = Postgresql
Trace                   = No
TraceFile               = /tmp/test_odbc.log
Database                = ets
Servername              = some.host
UserName                = paul
Password                = mySecret
Port                    = 5432
Protocol                = 6.4
ReadOnly                = No
RowVersioning           = No
ShowSystemTables        = No
ShowOidColumn           = No
FakeOidIndex            = No
ConnSettings            =
```

The above depends on the driver tag "Postgresql" being defined in the file /etc/odbcinst.ini, to give the actual driver file location. That file might have

something like

```
[PostgreSQL]
Description = PostgreSQL ODBC driver (Unicode version)
Driver          = /usr/lib/x86_64-linux-gnu/odbc/psqlodbcw.so
Setup           = /usr/lib/x86_64-linux-gnu/odbc/libodbcpsqlS.so
Debug = 0
CommLog = 1
UsageCount = 1
```

8.5 TSOracle Connection Details

This package is available on R-forge, but is not being tested, because I do not currently have a server to test it. The code in this section of the vignette is not being run. Please contact the package maintainer (Paul Gilbert) if you have an Oracle server and are willing to test the package.

The Oracle user, password, and hostname should be set in Oracle client configuration file (tnsnames.ora) before starting R.

The next small section of code uses *dbConnect* to set up database tables that expected by *TSconnect*.

```
> library("TSOracle")
> con  <- dbConnect("Oracle", dbname="test")
> source(system.file("TSsql/CreateTables.TSsql", package = "TSdbi"))
> dbDisconnect(con)
```

Now a TSdbi connection to the database is established.

```
>   con  <-  TSconnect("Oracle", dbname="test")
```

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>   detach(package:TSOracle)
>   detach(package:ROracle)
```

8.6 TSjson Connection Details

The *TSjson* method *TSconnect* can establish a connection to a proxy server. (See the main text for directly connecting to the web data source.)

```
> library("TSjson")
> con  <-  TSconnect("TSjson", dbname="proxy-cansim")
```

The *dbname* specifies the proxy server, for which credentials will be needed. The *user*, *password*, and *host*, can be specified as arguments. If specified as *NULL* (the default) then they will be determined by reading a file */.TSjson.cfg* which should have a line with four fields:

[proxy-cansim] user password host

The first field should match the *dbname* specification. Currently only a single line is supported, starting with "[proxy-cansim]", but the format is intended for extension to support proxies to different web databases.

If the file does not exist then environment variables "TSJSONUSER", "TSJSONPASSWORD", and "TSJSONHOST" will be used.

```
>   detach(package:TSjson)
>   detach(package:RJSONIO)
```

8.7 TSfame Connection Details

I no longer have access to Fame so package *TSfame* is no longer being extensively tested. The code in this section of the vignette is not being run. Please contact the package maintainer (Paul Gilbert) if you have Fame and are willing to test the package.

Beware that the package *fame* may be installed but not functional because the Fame HLI code is not available. A warning will be issued in this case.

Two variants of the Fame connect are available. The first requires a Fame database available on the local system:

```
>   con <- TSconnect("fame", dbname="testFame.db")
```

The second requires a Fame server:

```
>   con <- TSconnect("fame", dbname="ets /path/to/etsmfacansim.db", "read")
```

where the characters before the space in the dbname string indicate the network name of the server, and the path after the string indicates where the server should find the database.

While it may not be necessary to *detach* packages, the following prevents warnings later about objects being masked:

```
>   detach(package:TSfame)
>   detach(package:fame)
```

9 Appendix B: Underlying Database Structure and Loading Data

More detailed description of the instructions for building the database tables is given in the vignette for the *TSdbi* package. Those instruction show how to build the database using database utilites rather than R, which might be the way a system administrator would build the database.

The database tables are shown in the Table below. The *Meta* table is used for storing meta data about series, such as a description and longer documentation, and also includes an indication of what table the series data is stored in. To retrieve series it is not necessary to know which table the series is in, since this can be found on the *Meta* table. Putting data on the database may require

Table 1: Data Tables

Table	Contents
Meta	meta data and index to series data tables
A	annual data
Q	quarterly data
M	monthly data
S	semiannual data
W	weekly data
D	daily data
B	business data
U	minutely data
I	irregular data with a date
T	irregular data with a date and time

specifying the table, if it cannot be determined from the R representation of the series.

In addition, there will be tables "vintages" and "panels" if those features are used.

The following is done with dbConnect in place of a TSconnect, since they are direct SQL queries and do not require the TSdbi structure.

The structure reported reflects the setup that was done previously. These queries are Mysql specific but below is a generic SQL way to do this.

```
> library("TSMySQL")
> con <- dbConnect("MySQL", dbname="test")
> dbListTables(con)

 [1] "A"    "B"    "D"    "I"    "M"    "Meta" "Q"    "S"    "T"    "U"
[11] "W"

> dbGetQuery(con, "show tables;")

   Tables_in_test
1               A
2               B
3               D
4               I
5               M
6            Meta
7               Q
8               S
9               T
10              U
11              W
```

```
>  dbGetQuery(con, "describe A;")

  Field        Type Null Key Default Extra
1    id varchar(40)  YES MUL    <NA>
2  year     int(11)  YES MUL    <NA>
3     v      double  YES        <NA>

>  dbGetQuery(con, "describe B;")

   Field        Type Null Key Default Extra
1     id varchar(40)  YES MUL    <NA>
2   date        date  YES MUL    <NA>
3 period     int(11)  YES MUL    <NA>
4      v      double  YES        <NA>

>  dbGetQuery(con, "describe D;")

   Field        Type Null Key Default Extra
1     id varchar(40)  YES MUL    <NA>
2   date        date  YES MUL    <NA>
3 period     int(11)  YES MUL    <NA>
4      v      double  YES        <NA>

>  dbGetQuery(con, "describe M;")

   Field        Type Null Key Default Extra
1     id varchar(40)  YES MUL    <NA>
2   year     int(11)  YES MUL    <NA>
3 period     int(11)  YES MUL    <NA>
4      v      double  YES        <NA>

>  dbGetQuery(con, "describe Meta;")

          Field        Type Null Key Default Extra
1            id varchar(40)   NO PRI    <NA>
2           tbl     char(1)  YES MUL    <NA>
3     refperiod varchar(10)  YES        <NA>
4   description        text  YES        <NA>
5 documentation        text  YES        <NA>

>  dbGetQuery(con, "describe U;")

   Field        Type Null Key           Default                         Extra
1     id varchar(40)  YES MUL              <NA>
2   date   timestamp   NO MUL CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP
3     tz  varchar(4)  YES                  <NA>
4 period     int(11)  YES MUL              <NA>
5      v      double  YES                  <NA>
```

```
>  dbGetQuery(con, "describe Q;")

   Field        Type Null Key Default Extra
1     id varchar(40)  YES MUL    <NA>
2   year     int(11)  YES MUL    <NA>
3 period     int(11)  YES MUL    <NA>
4      v      double  YES        <NA>

>  dbGetQuery(con, "describe S;")

   Field        Type Null Key Default Extra
1     id varchar(40)  YES MUL    <NA>
2   year     int(11)  YES MUL    <NA>
3 period     int(11)  YES MUL    <NA>
4      v      double  YES        <NA>

>  dbGetQuery(con, "describe W;")

   Field        Type Null Key Default Extra
1     id varchar(40)  YES MUL    <NA>
2   date        date  YES MUL    <NA>
3 period     int(11)  YES MUL    <NA>
4      v      double  YES        <NA>
```

If schema queries are supported then table information can be obtained in a (almost) generic SQL way. On some systems this will fail because users do not have read priveleges on the INFORMATION_SCHEMA table. This does not seem to be an issue in SQLite, but SQLite schema queries are not the same as for other SQL engines.

```
>   dbGetQuery(con, paste(
         "SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.Columns ",
         " WHERE TABLE_SCHEMA='test' AND table_name='A' ;"))

  COLUMN_NAME
1          id
2        year
3           v

>   dbGetQuery(con, paste(
         "SELECT COLUMN_NAME, COLUMN_DEFAULT, COLLATION_NAME, DATA_TYPE,",
         "CHARACTER_SET_NAME, CHARACTER_MAXIMUM_LENGTH, NUMERIC_PRECISION",
   "FROM INFORMATION_SCHEMA.Columns WHERE TABLE_SCHEMA='test' AND table_name='A' ;"))

  COLUMN_NAME COLUMN_DEFAULT    COLLATION_NAME DATA_TYPE CHARACTER_SET_NAME
1          id           <NA> latin1_swedish_ci   varchar             latin1
2        year           <NA>              <NA>       int               <NA>
3           v           <NA>              <NA>    double               <NA>
```

```
  CHARACTER_MAXIMUM_LENGTH NUMERIC_PRECISION
1                       40                NA
2                       NA                10
3                       NA                22
```

```
>   dbGetQuery(con, paste(
     "SELECT COLUMN_NAME, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH, NUMERIC_PRECISION",
   "FROM INFORMATION_SCHEMA.Columns WHERE TABLE_SCHEMA='test' AND table_name='M';"))
```

```
  COLUMN_NAME DATA_TYPE CHARACTER_MAXIMUM_LENGTH NUMERIC_PRECISION
1          id   varchar                       40                NA
2        year       int                       NA                10
3      period       int                       NA                10
4           v    double                       NA                22
```

```
>   dbDisconnect(con)
```

```
[1] TRUE
```

10 Appendix C: Examples Using DBI and direct SQL Queries

The following examples are queries using the underlying "DBI" functions. They should not often be needed to access time series, but may be useful to get at more detailed information, or formulate special queries. Typically these queries may be more useful for systems administrators doing database maintenance than they are for end users.

These queries depend on the underlying structure of the database, which should be considered "opague" from the perspective of a TSdbi user. That is, this structure could be changed without affecting the TSdbi functionality, but the following queries would be affected.

```
>  library("TSMySQL")
>  con <- TSconnect("MySQL", dbname="test")
>  dbGetQuery(con, "SELECT count(*) FROM Meta ;")
```

```
  count(*)
1        0
```

```
>  dbGetQuery(con, "SELECT max(year) FROM A ;")
```

```
  max(year)
1        NA
```

Finally, to disconnect gracefully, one should

```
> dbDisconnect(con)
```