

Bannert, Matthias

Working Paper

timeseriesdb: Manage and archive time series data in establishment statistics with R and PostgreSQL

KOF Working Papers, No. 384

Provided in Cooperation with:

KOF Swiss Economic Institute, ETH Zurich

Suggested Citation: Bannert, Matthias (2015) : timeseriesdb: Manage and archive time series data in establishment statistics with R and PostgreSQL, KOF Working Papers, No. 384, ETH Zurich, KOF Swiss Economic Institute, Zurich,
<https://doi.org/10.3929/ethz-a-010480183>

This Version is available at:

<https://hdl.handle.net/10419/114458>

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

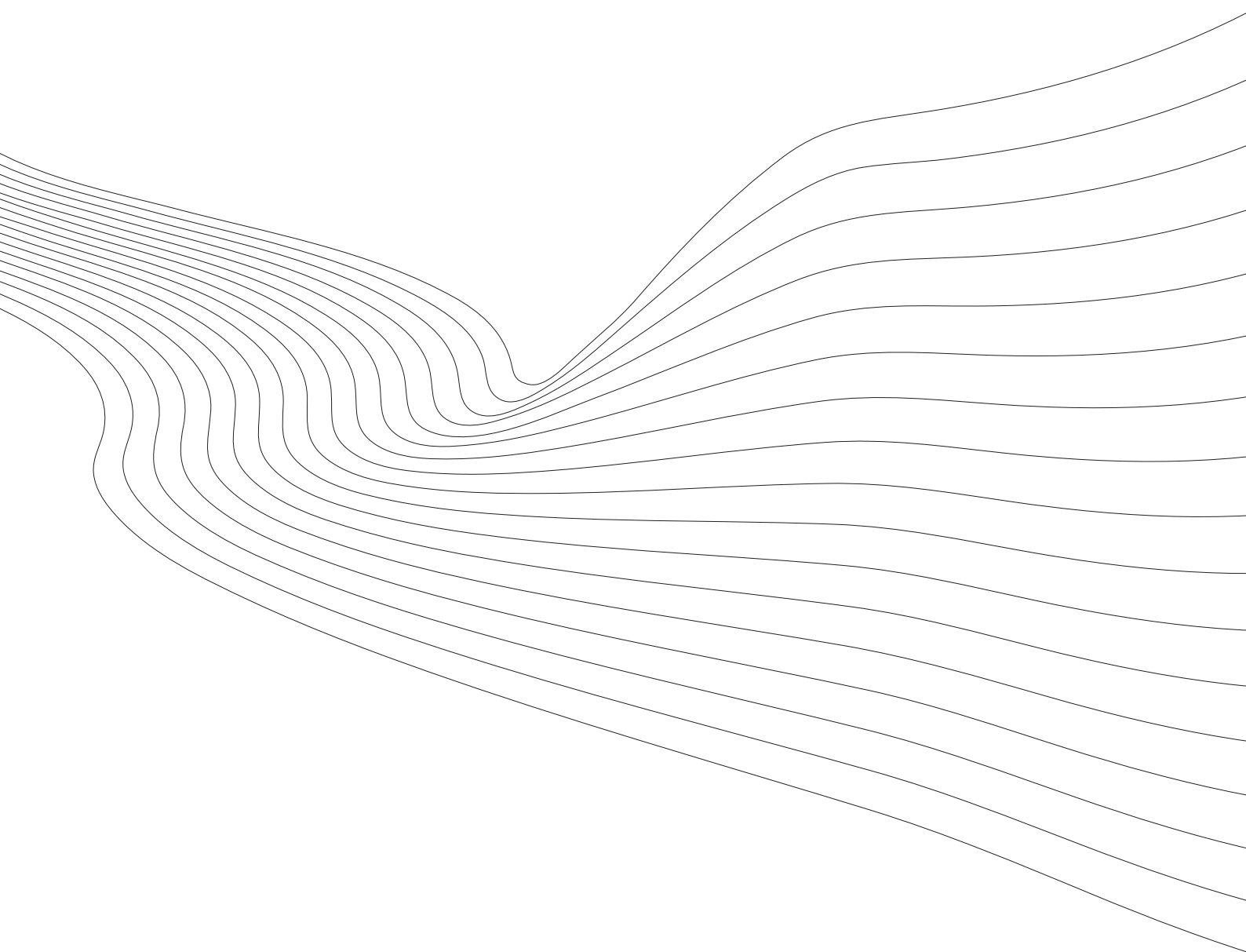
You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.

KOF Working Papers

timeseriesdb: Manage and Archive Time Series Data in
Establishment Statistics with R and PostgreSQL

Matthias Bannert



KOF

KOF Swiss Economic Institute
ETH Zurich
LEE G 116
Leonhardstrasse 21
8092 Zurich
Switzerland

Phone +41 44 632 42 39
Fax +41 44 632 12 18
www.kof.ethz.ch
kof@kof.ethz.ch

timeseriesdb: Manage and Archive Time Series Data in Establishment Statistics with R and PostgreSQL

Matthias Bannert

KOF Swiss Economic Institute, ETH Zurich

Abstract

timeseriesdb is an R package which suggests a PostgreSQL database structure to store time series alongside extensive multi-lingual meta information and provides an R database interface including a web based GUI. The **timeseriesdb** package was designed to handle time series in establishment statistics. Information such as the GDP or data stemming from the aggregation of economic surveys is typically published on a monthly, quarterly or yearly basis. Hence the package is optimized to handle a large amount of different time series as opposed to managing a smaller number of high frequency time series such as real time data obtained from measuring devices. The particular focus of **timeseriesdb** is to help the user find and extract a particular set of information within a larger set of information. The **timeseriesdb** package intends to provide the infrastructure for a time series catalog as opposed to handling time series operations on database level. The underlying structure relies on PostgreSQL's *hstore* data type which allows to store an array of key-value pairs in a single cell. The *hstore* data type is not only used to reduce the number of records by storing an entire time series in a single record but also to store a record specific amount of multi-lingual meta information items flexibly.

Keywords: time series, data management, relational database, establishment statistics, official statistics, hstore, NoSQL, economic data, reproducible research, R, PostgreSQL.

1. Introduction¹

Time series data are used in a plethora of research fields from ecology to econometrics. Thus the R Language for Statistical Computing provides a broad variety of time series related functionality already in its basic setup. In addition, a large number of extension packages can be downloaded from CRAN to meet heterogenous demands. Also, the R community provides a CRAN Task View (CTV) entirely dedicated to time series². While most packages are designed to handle times and dates, seasonality, stationarity, unit roots, cointegration, define time series classes, do forecasting and modelling, frequency analysis, decomposition and filtering or resampling, packages that actually aim at archiving time series are rather scarce. This can be regarded as a shortcoming of the R ecosystem so far as archiving particular versions of time series can be crucial for reproducibility of publications.

Though it is suitable for most researchers to store time series in R's own *.RData* format or to use standard flat files formats like *.csv* or *.dif* it is not sufficient to manage larger archives of several hundred million time series. Yet already at a much smaller amount of time series researchers can profit from storing and managing data in a relational database. Databases are designed to find a particular set of information within a larger set of information. Also relational databases provide the opportunity to add further information that relates to one or more sets of information. In order to effectively make use of information stored in a relational database from inside an R session the common database interface R package **DBI** (Databases 2014) can be used. A connection to directly query the database can then be set up using the database management system (DBMS) specific package, e.g. **Rpostgresql** (Conway et al. 2013) or **ROracle** (Denis Mukhin and Luciani 2014).

However, these basic interfaces do not provide a higher level interface in the sense that they map R classes such as *ts* for time series to the database tables and vice versa. Rather these packages can be used to write SQL queries as character strings and send them as queries to the database. Typically, results are returned as standard R *data.frames*. Built on top of the interface described above, the pioneering **TSdbi** (Gilbert 2013a) package along with a family of corresponding DBMS specific packages such as **TSMysql** (Gilbert 2013b) or **TSPostgreSQL** (Gilbert 2013c) have addressed the need to conveniently map R time series objects to relations in a database.

This paper introduces **timeseriesdb** (Bannert 2015) and suggests an alternative approach to store and manage time series data in a relational database. The **timeseriesdb** package distinguishes itself from existing packages by two factors in particular: First, an entire time series is stored in a single table cell as opposed to storing one record per observation. This reduces the number of records substantially, particularly for high frequency data and long time series. Second, **timeseriesdb** enables users to store extensive meta information in multiple

¹I would like to thank Jan-Egbert Sturm, Ulf-Dietrich Reips and Klaus Abberger for their great support of my work in between the fields of economics, psychology, statistics and software development. I also would like to thank the participants of the Webdatanet Conference for helpful remarks as well my colleagues Andreas Dibiasi, Dirk Drechsel and Pauliina Sandqvist for fruitful discussions, their testing efforts and feedback. I am thankful to Ioan Gabriel Bucur and Charles Clavadetscher for their valuable contributions and code reviews. I would also like to thank Craig Ringer for his insights on concurrent database inserts. Further I am thankful to the participants of the Webdatanet Conference in Salamanca for their valuable comments. This working paper version uses a modified version of the JSS Rmarkdown template by Achim Zeileis.

²CTVs monitor, describe and summarizes R packages in a particular field. The CTV for time series analysis is maintained by Rob J. Hyndman and can be found at <http://cran.r-project.org/web/views/TimeSeries.html>.

languages. Also the amount of translated meta information items may vary from record to record. Furthermore, **timeseriesdb** strives to optimize query time when reading from the database or writing to it.

The subsequent chapter continues to further motivate the use **timeseriesdb** particularly in context of reproducible research and archiving data alongside corresponding meta data. The remainder of this paper is structured as follows: The third chapter covers data storage in greater detail and elaborates on the relational structure of the underlying PostgreSQL database. The fourth chapter gives an overview of the mapping functionality of **timeseriesdb** and shows how R objects are mapped to database relations. Chapter five provides applied code examples to illustrate basic features such as reading and writing from the database. Besides the web-based graphical user interface is introduced. Finally the paper is concluded and an outlook to future releases is given.

2. Motivation

Originally designed to archive economic survey data on the aggregated level, **timeseriesdb** is well suited to handle any kind of official statistics time series with a monthly, quarterly or yearly release cycle. Establishment statistics are often shared on the aggregated level only, because micro level data – i.e. company or household level data is sensitive. If data is not shared at the micro level and time series cannot be reproduced by the recipient, proper meta information becomes particularly crucial. Against the background of the ongoing development in reproducible research (Koenker and Zeileis 2009) the ability to trace data back to their provider and to describe data extensively becomes a central aspect of an empirical researcher’s work³. McCullough and Vinod (2003) check five years of American Economic Review articles that involve non-linear solvers for their authors’ efforts to verify solutions provided by their respective software package. Despite the fact that previous research (Stokes 2004) had pointed out issues to reproduce even top contributions of the profession that involve non-linear solvers, McCullough and Vinod (2003) does not find a single author who reports some kind of efforts to verify results produced by statistical software that involves non-linear solvers. Consequently McCullough and Vinod (2003) claims that policies yet alone are not sufficient and only both, data and code archives can ensure reproducibility of empirical economic research. When reproduction problems occur at a later stage solid code and data archives are mandatory to trace back results and circumstances of the data generating process.

McCullough (2009) shows an overview of economic journals that have made data and code archives mandatory since 1990. By today, the list includes top journals of the economic profession like the American Economic Review (AER) which introduced their archive ten years ago in 2004. Also more recently *Reproducible Research* has experienced yet another push with the R community at the forefront: particularly the **knitr** package (Xie 2015), (Xie 2013), (Xie 2014) and **rmarkdown** (Allaire et al. 2014) are widely used to create documents dynamically from text, code and data. Gandrud (2013) provides a summary on creating dynamic documents with R including application examples. However, with a reproducible approach to generate reports and research papers, data descriptions can also be loaded dynamically if comprehensive meta information is available. In this context of archiving data and reproducing

³Buckheit and Donoho date the origin of the reproducible research movement back to Stanford geophysicist Jon Clearbout, quoting his claim that the scientific publication is not the scholarship but rather its advertising. Consequently he claims that software is the actual scholarship in a computing driven research project.

research documents Leeper (2014) expresses the need for services that offer data with sophisticated meta data support. Hence the motivation to create **timeseriesdb** came in large parts from the intention to design a time series archive that allows to store comprehensive meta information and make this information available in context of a dynamic process such a creating reproducible documents with **knitr** or **rmarkdown**. Also, multi-lingual meta information should help to find data and thus foster global exchange of time series information. This meta information also becomes important when storing time series information that potentially has multiple versions (vintages). National Accounts and GDP time series for example are subject to revisions (Shrestha and Marini 2013),(Branchi et al. 2007) and thus it is important to know which version of time series was used in research or forecasting exercises. The ability of **timeseriesdb** to store extensive meta information and make it available in the context of the computational process helps to avoid confusion of series as users can label and select series dynamically. With its license cost free open source components, light weight architecture and low maintenance costs **timeseriesdb** is designed to also suit smaller data providers⁴.

Another important aspect that motivated the development of **timeseriesdb** is its explicit focus on archiving. While other approaches such as druid (Yang et al.) often execute adhoc computation on database level at query time, **timeseriesdb** leaves time series operations to R and packages that were explicitly designed to process time series. Time series in official statistics can often be the result of complex or computationally intensive processes such as hierarchical aggregation schemes or seasonal adjustment which use specific software. Seasonal adjustment for example, is most often done with the U.S. Census Bureau's X13-ARIMA-SEATS Fortran program. The **seasonal** package (Sax 2014) provides the opportunity to use the Census Bureau's software from within R. Furthermore R reads numerous foreign file formats, e.g. using **foreign** (R Core Team 2015) or **R.matlab** (Bengtsson 2015) and offers interfaces to various standard software packages as well as to domain specific software. Thus R can be used as a flexible interface between other software and the archive database suggested by **timeseriesdb**. In other words **timeseriesdb** explicitly focuses on storing pre-computed time series and providing a data catalog while avoiding the need for comprehensive SQL knowledge from the user.

⁴Please find installation and setup instruction for the database in the appendix of this paper.

3. Data Storage

This chapter covers the relational structure of the PostgreSQL database that is used in **timeseriesdb**. Figure 1 provides an overview of all tables and views used in the database schema. The following sub chapters explain the details of storing time series records as well as corresponding meta information.

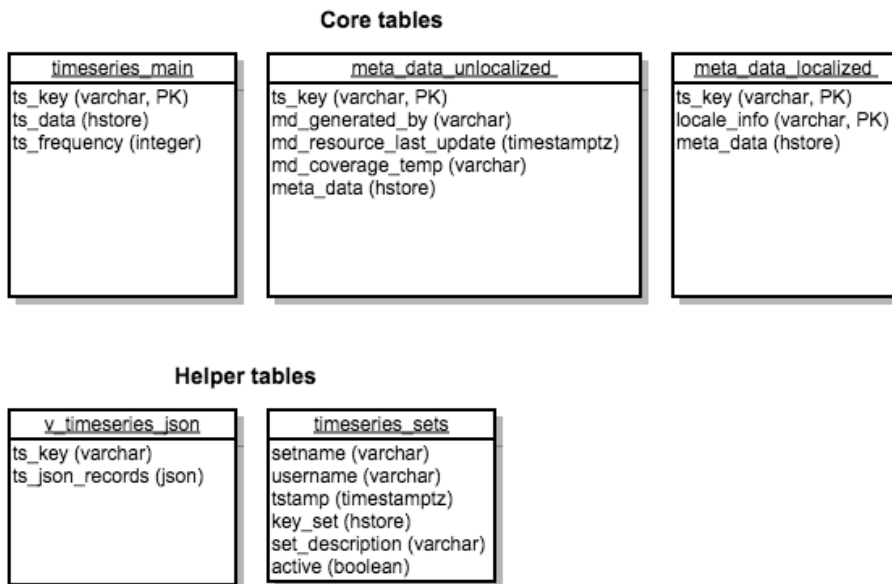


Figure 1: Overview: Relational Structure

3.1. Data: Time Series Records

The time series records are stored in a simple three column table called *timeseries_main*. While the unique identifier of the record as well as the frequency of the time series are stored as standard *varchar* and *integer* data types respectively, the time series are stored in the PostgreSQL specific *hstore* format. The *hstore* data type stores key-value pairs in a single table cell. In the case of time series the date is regarded as the key while the actual value represents the value of the pair. By using *hstore* an entire time series can be stored in a single row and does not need to be stored in an one-row-per-observation format. PostgreSQL provides functionality to access *hstore* keys inside a row to extract specific parts of the series. In order to facilitate access **timeseriesdb** provides a view⁵ called *v_timeseries_json* which casts data into the popular, more standard *JSON*⁶ format. *JSON* allows for nested structures and thus an entire time series record, that is key, frequency and time series data itself, can be cast into a single *JSON* entry. With the help of this *JSON* based view reading time series into R could be speeded up substantially compared to using multiple select statements or splitting results of a single call on the R level⁷. On the R level **timeseriesdb** makes use of **RJSONIO**

⁵Views are basically stored select statements in SQL.

⁶JavaScript Object Notation.

⁷see Appendix 'benchmarks' for a detailed summary on the reading speed.

(Lang 2014) to process *JSON*.

3.2. Meta Data: Localized and Unlocalized information

While the *hstore* format implements non-relational concepts inside a relational database, meta information is stored in separate relations and linked to the main records. **timeseriesdb** uses two tables to store meta information: the table *meta_information_unlocalized* is structured to store meta information that is not translated such as usernames or time span of coverage. The second meta information table, namely *meta_information_localized* holds meta information that could be translated. Information such as wording of questions in survey based time series or elaborate descriptions are stored in the latter table. Again the *hstore* format is used in both tables to allow for a flexible amount of meta information for each record. For example some record may have a meta description in French, German and English while another record only contains German and English meta information. Using *hstore* this situation can be covered in a single table without storing empty values.

3.3. Storing Sets of Time Series

The second helper table called *timeseries_sets* allows users to store a set of time series keys under a setname (*varchar*). This can be helpful when users return regularly to retrieve updates of the same series at a later stage. The table *timeseries_sets* will also store the username as a *varchar* and the current timestamp at the time of storage. The set of keys itself is stored in the *hstore* format with the time series key being the key and the type of key being the value. This allows to use other keys than the primary time series key, e.g. meta information, to identify the time series that belong to a set. In addition a set_description can be added as *varchar* and the active flag can be used to deactivate respectively activate an existing set.

4. Mapping SQL relations to R objects

The focal functionality of **timeseriesdb** is to map R objects to database relations and vice versa. In general time series are identified by an unique primary key. Hence this chapter introduces the most important functions to perform the object - relational mapping between R and PostgreSQL based on identification by an unique time series key⁸:

```
readTimeSeries()
readMetaInformation()
storeTimeSeries()
storeMetaInformation()
createMetaDataHandle()
createHstore()
```

4.1. Function Overview

As its name suggests, the first function reads the time series itself from the database by its key and returns a list that contains at least one R object of class *ts*. The function *readMetaIn-*

⁸For more detailed information see the code examples in the next chapter.

formation reads meta information from the database given unique identification of time series and returns an environment which contains meta information. This environment holds meta information objects named like the corresponding time series records. By using a separate environment objects can have the same name as the corresponding time series and are thus easy to link. Also the resulting meta information environment can be updated by further calls of *readMetaInformation* at a later stage.

Similarly *storeTimeSeries* stores time series that reside in an R environment or list to the database. Again, a character vector containing time series names can be passed to *storeTimeSeries*. The function *storeMetaInformation* works analogously to the *readMetaInformation* as it reads meta information from a specified environment and stores them to the database.

The second but last in the list of functions presented above is less essential but provides an easy way to create convenience operators. These operators help to look up the database using regular expressions. The code chunk below shows a general example of a pre-defined operator that works with the main tables fixed primary key *ts_key*.

```
con %k% 'ts[1-9]{1}$'
```

The object *con* is a PostgreSQL connection object and character string is a regular expression pattern. In this example the pattern matches all keys $\in ts1, \dots, ts9$. The *%k%* operator searches the main tables keys for all records that fit the expression and returns a list of time series that matched the query. Such an operator is easy to pre-define because the primary key is fixed and typically not changed by the user. However, querying meta information is very different because users define the amount of meta information keys and their names. Thus **timeseriesdb** provides a function to create such operators for flexible keys.

```
"%lk%" <- createMetaDataHandle("legacy_key")
con %lk% 'old_key[1-9]{1}$'
```

The example above assumes that there is a meta information field for legacy keys. This means that at least one *hstore* record contains a key called *legacy_key*. After the corresponding operator is created, the operator can be used just like the operator in the fixed key example presented above. Note that, the operators created by *createMetaDataHandle* reside in the global environment as opposed to the namespace of the package and are thus removed when the global environment is cleared, e.g. by *rm(list=ls())*. The last function in the list, namely *createHstore* is a method typically not used by the user but is a helper function which is crucial and often used in the mapping process. The function uses the PostgreSQL function *hstore* to create key value pairs from different types of R objects. As of version v0.21 of **timeseriesdb** methods for classes *ts*, *data.frame* and *list* exists. It is important to understand that *hstore* does not understand nested formats. Hence the input is limited to simple, unnested *data.frames* and *lists*

5. Code Examples

The core functionality of **timeseriesdb** is to read and write time series data and their corresponding meta information into a PostgreSQL database using R. The following examples illustrate this basic functionality step by step and continue to describe convenience functions

such as plotting a list of time series. For the subsequent examples to work a database connection to a database that contains a schema called *timeseries* with the relations suggested by **timeseriesdb** is needed. SQL statements for a first time set up of all necessary relations, functions and triggers can be found in the *inst/sql* folder of **timeseriesdb**⁹.

The database connectivity of **timeseriesdb** depends on the R packages **DBI** (Databases 2014) and **RPostgreSQL** (Conway et al. 2013)¹⁰. The **timeseriesdb** package comes with a convenience function *createConObj* to create a PostgreSQL connection object. The *createConObj* function expects several connection parameters which can be either directly passed to the function or stored as constants using *Sys.setenv*. The advantage of using the system environment as opposed to any object in R's global environment is that is not affected by clearance of session memory. If the database connection does not change regularly it can be attractive to store the host name, database name and schema either in a global or user specific *.Renviron* file. User *.Renviron* files are typically located in the user's home directory.

```
library(timeseriesdb)

# set database name
Sys.setenv(TIMESERIESDB_NAME = "sandbox")
Sys.setenv(TIMESERIESDB_HOST = "localhost")
Sys.setenv(TIMESERIESDB_SCHEMA = "timeseries")
con <- createConObj(passwd = "")
```

The local example database used in this paper does not use a password. Obviously this is not suitable for most production use cases. If password protection is desired, entering passwords interactively should be preferred over storing password in text files. *R Studio* provides the opportunity to use its interface to make use of an interactive passwords prompt that hides the password from the screen. Figure 2 shows *R Studio*'s password dialogue. Also note that the user is taken from *Sys.info* and needs to be specified separately as an argument of *createConObj* if the database user deviates from the system user.

⁹Further installation notes can be found in the appendix of this paper.

¹⁰We will closely monitor the development of **RPostgres** which is currently being developed by Hadley Wickham, but is not an official CRAN package yet. It also relies on **DBI** but has a different design approach. The **Rpostgres** packages might be an alternative to implement database connectivity or could also be supported additionally.

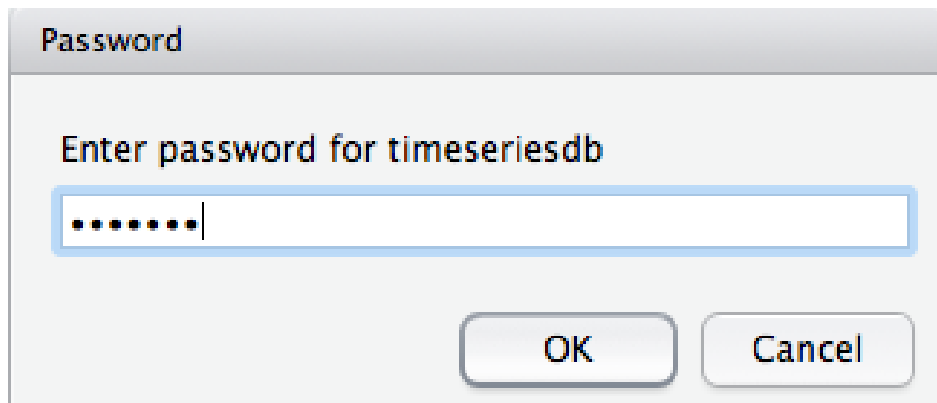


Figure 2: R Studio Password Prompt

5.1. Write Time Series to Database

In order to demonstrate writing to the database, a set of 100 random time series is created first. The created time series are of R's standard time series class *ts*. All time series are generated using a random normal and resulting series are stored in a list. List elements are named with a unique, sequential identifier.

```
# set a seed to make
# the result fully reproducible
set.seed(123)
nms <- paste0("ts",1:100)
ts_list <- lapply(nms, function(x) ts(rnorm(50),
                                     start = c(1998,1),
                                     frequency = 4))

class(ts_list[[1]])

[1] "ts"

names(ts_list) <- nms
```

The function to actually store R time series objects to the database is *storeTimeSeries*. It takes names of time series as arguments and searches for corresponding time series in an environment or list. For performance reasons the function *storeTimeSeries* has been optimized to only use one single SELECT statement to store multiple series. Thus the user should avoid looping over *storeTimeSeries* respectively not use *apply* family functions with *storeTimeSeries*. Rather a vector of time series names should be passed to the *storeTimeSeries* when multiple series should be stored.

```
storeTimeSeries(nms,con,li = ts_list)

[1] "100 data and meta data records written successfully."
```

As the output from `storeTimeSeries` shows, information is stored to the main time series table as well as to meta information table. When a time series record is stored, a minimal amount of meta data is stored to the unlocalized meta information table. These data are user information, time and timespan covered by the time series. All of this information can be derived from the storage process or the time series itself. Figure 3 shows a PostgreSQL client window with a query on the main time series table displaying one of the test time series. Figure 4 shows the corresponding minimal meta information that is generated with the initial storage process.

```

ts_key      | ts1
ts_data     | "1998-01-01"=>"0.922301059107954", "1998-04-01"=>"-2.45149100964402", "199
8-07-01"=>"-0.131003819224936", "1998-10-01"=>"-1.05339701036674", "1999-01-01"=>"1.12716
590131421", "1999-04-01"=>"-0.727834639228732", "1999-07-01"=>"0.935340589208178", "1999-
10-01"=>"-0.468292103835751", "2000-01-01"=>"0.12982106876817", "2000-04-01"=>"1.46235284
119442", "2000-07-01"=>"-0.682169375643072", "2000-10-01"=>"1.81861838674073", "2001-01-0
1"=>"0.986158368247289", "2001-04-01"=>"1.28460132176685", "2001-07-01"=>"-2.246400569400
27", "2001-10-01"=>"-0.168516631588257", "2002-01-01"=>"-1.46661662751573", "2002-04-01"=
>"0.759275035989203", "2002-07-01"=>"1.2227702922995", "2002-10-01"=>"-0.617535389141603
", "2003-01-01"=>"-0.511773942223519", "2003-04-01"=>"-1.62158019298443", "2003-07-01"=>"
0.790937635334595", "2003-10-01"=>"1.46152195521853", "2004-01-01"=>"-1.69993221571343",
"2004-04-01"=>"-1.81251474524571", "2004-07-01"=>"1.14414109742141", "2004-10-01"=>"1.348
54185723644", "2005-01-01"=>"0.371556455073257", "2005-04-01"=>"0.242249032302166", "2005
-07-01"=>"-0.621258548098154", "2005-10-01"=>"0.339038070553804", "2006-01-01"=>"-0.45214
0126031672", "2006-04-01"=>"2.04323321252589", "2006-07-01"=>"-0.449337685075739", "2006-
10-01"=>"-3.13738452799538", "2007-01-01"=>"0.499962209076086", "2007-04-01"=>"-1.2571415
8968218", "2007-07-01"=>"0.822761428159943", "2007-10-01"=>"-1.54609608135241", "2008-01-
01"=>"-0.258780758988273", "2008-04-01"=>"0.390407377124082", "2008-07-01"=>"-0.197270203
079809", "2008-10-01"=>"-1.94694947947191", "2009-01-01"=>"-1.42763817341705", "2009-04-0
1"=>"-0.850418042257596", "2009-07-01"=>"1.62446909297786", "2009-10-01"=>"-0.12663816139
8654", "2010-01-01"=>"1.27560202747805", "2010-04-01"=>"0.179496176715821"
ts_frequency | 4
sandbox=#

```

Figure 3: PostgreSQL Client: A Time Series Record

```

ts_key      | ts1
ts_data     | "1998-01-01"=>"0.922301059107954", "1998-04-01"=>"-2.45149100964402", "199
8-07-01"=>"-0.131003819224936", "1998-10-01"=>"-1.05339701036674", "1999-01-01"=>"1.12716
590131421", "1999-04-01"=>"-0.727834639228732", "1999-07-01"=>"0.935340589208178", "1999-
10-01"=>"-0.468292103835751", "2000-01-01"=>"0.12982106876817", "2000-04-01"=>"1.46235284
119442", "2000-07-01"=>"-0.682169375643072", "2000-10-01"=>"1.81861838674073", "2001-01-0
1"=>"0.986158368247289", "2001-04-01"=>"1.28460132176685", "2001-07-01"=>"-2.246400569400
27", "2001-10-01"=>"-0.168516631588257", "2002-01-01"=>"-1.46661662751573", "2002-04-01"=
>"0.759275035989203", "2002-07-01"=>"1.2227702922995", "2002-10-01"=>"-0.617535389141603
", "2003-01-01"=>"-0.511773942223519", "2003-04-01"=>"-1.62158019298443", "2003-07-01"=>"
0.790937635334595", "2003-10-01"=>"1.46152195521853", "2004-01-01"=>"-1.69993221571343",
"2004-04-01"=>"-1.81251474524571", "2004-07-01"=>"1.14414109742141", "2004-10-01"=>"1.348
54185723644", "2005-01-01"=>"0.371556455073257", "2005-04-01"=>"0.242249032302166", "2005
-07-01"=>"-0.621258548098154", "2005-10-01"=>"0.339038070553804", "2006-01-01"=>"-0.45214
0126031672", "2006-04-01"=>"2.04323321252589", "2006-07-01"=>"-0.449337685075739", "2006-
10-01"=>"-3.13738452799538", "2007-01-01"=>"0.499962209076086", "2007-04-01"=>"-1.2571415
8968218", "2007-07-01"=>"0.822761428159943", "2007-10-01"=>"-1.54609608135241", "2008-01-
01"=>"-0.258780758988273", "2008-04-01"=>"0.390407377124082", "2008-07-01"=>"-0.197270203
079809", "2008-10-01"=>"-1.94694947947191", "2009-01-01"=>"-1.42763817341705", "2009-04-0
1"=>"-0.850418042257596", "2009-07-01"=>"1.62446909297786", "2009-10-01"=>"-0.12663816139
8654", "2010-01-01"=>"1.27560202747805", "2010-04-01"=>"0.179496176715821"
ts_frequency | 4
sandbox=#

```

Figure 4: PostgreSQL Client: A Meta Information Record

5.2. Read Time Series From the Database

The corresponding function to read time series from the database is called *readTimeSeries*. Analogously to *storeTimeSeries*, *readTimeSeries* takes a vector of time series keys and a connection object as its minimal arguments¹¹.

```
# clear the memory
rm(list=ls())
con <- createConObj(passwd = "")
results <- readTimeSeries(c("ts1","ts2","ts3"),con)
# note the class of results
class(results)
```

```
[1] "list"  "tslist"
```

Note that *readTimeSeries* always returns a list of time series, no matter how many elements different from zero (i.e. time series) are actually returned. Again, *readTimeSeries* has been optimized for bulk loading a large amount of time series from the database. Hence multiple SELECT statements are avoided by the function as it allows to process a vector of time series keys directly. In turn the user should not use *readTimeSeries* in loops or *apply* constructions¹².

5.3. Adding and Reading Elaborate Meta Information

Meta information can either be added on data base level or using R. When adding comprehensive meta information **timeseriesdb** distinguishes between meta information that cannot be translated (e.g. username or timestamps) and meta information that is typically translated (e.g. a spoken meta description). In both cases **timeseriesdb** uses the flexible *hstore* format again. The use of *hstore* enables **timeseriesdb** to store a different amount of meta information for every record without having to create empty data cells in a rectangular data format for those records that do not have a particular type of meta information.

While a minimal amount of meta information has been stored when a time series record itself was stored, users can add further localized and unlocalized meta information to the database. First the code chunk below shows how to add additional meta information without a specific locale The basic idea is to generate a separate environment that holds all unlocalized meta information while the time series may reside in the *.GlobalEnv* or any other environment distinct from the meta environment. Meta information is meant to have the same object name as the corresponding time series.

```
# add seed info that was
# used to create the series and some legacy id
meta_ts1 <- list(seed = 123,legacy_key = 'series1')
meta_ts2 <- list(seed = 123,legacy_key = 'series2')
```

¹¹This document has been created using the **knitr** (Xie, 2015) package for reproducible research. All code chunks in this paper are executed during compile of the document. Thus objects are removed from R's session memory to truly illustrate interaction with the database.

¹²For a more detailed insight on query performance, see Appendix: Query Benchmarks.

```
meta_unlocalized <- addMetaInformation("ts1",meta_ts1)
addMetaInformation("ts2",meta_ts2,meta_unlocalized)
```

`x` contains 2 meta information object(s).

The same function can be used in analogous fashion to create an environment that holds localized meta information. The following example adds English meta information to the time series `ts1`. Note that the `locale` argument of `addMetaInformation` defaults to `NULL` and thus only needs to be set when localized meta information is added.

```
en <- list('short_description' = 'Random Series',
          'full_description' = 'Random Time Series generated
                               from a Standard Normal using seed 123.'
          )
```

```
meta_en <- addMetaInformation('ts1',en)
meta_en$ts1
```

```
$short_description
[1] "Random Series"
```

```
$full_description
[1] "Random Time Series generatedrom a Standard Normal using seed 123."
```

```
attr("class")
[1] "miro" "list"
```

In both cases meta information is stored in separate R environments. In a second step we can store all records contained in these environments to the database as shown in the following code chunk.

```
# store localized Information
storeMetaInformation("ts1",con,locale = "en",
                    lookup_env = "meta_en",overwrite = T)
```

```
[1] "en meta information successfully written."
```

```
# store unlocalized information
storeMetaInformation("ts1",con,tbl = "meta_data_unlocalized",
                    "meta_unlocalized",
                    locale = NULL,
                    overwrite = T)
```

```
[1] " meta information successfully written."
```

Similarly, meta information can be read from the database into an R session. Loading meta information is done explicitly because meta information should not be implicitly loaded into the R session when reading in time series. Also, users should be able to vary the amount of meta information that is loaded into context. Depending on the different use cases such as batch processing or exploratory data research a different amount of meta information may be desired. Thus reading meta information is separated from reading data.

```
m_unlocal <- readMetaInformation('ts2',con,NULL,tbl = "meta_data_unlocalized")
m_local <- readMetaInformation('ts1',con,"en")
m_unlocal$ts1 ##
```

```
NULL
```

```
m_local$ts1 ##
```

```
$full_description
[1] "Random Time Series generated from a Standard Normal using seed 123."
```

```
$short_description
[1] "Random Series"
```

```
attr(,"class")
[1] "miro" "list"
```

Meta information can be used to label figures and tables dynamically. Figure 5 shows a time series plot with a dynamically created title loaded from a PostgreSQL database.

```
plot(results$ts1, main = m_local$ts1$short_description)
```

5.4. Plot a List of Time Series

The **timeseriesdb** package will return a list of time series to the R session when the database is queried for time series records. Because base R does not provide a function to plot a list of time series out of the box, **timeseriesdb** comes with a convenience method that accepts a list with the additional class ‘tlist’ containing one or more elements of class “ts”. Thus **timeseriesdb** functions that return lists of time series, namely the *readTimeSeries* as well as the quick handle operators presented in chapter 4, add the class attribute ‘tlist’ to the returned list. Thus the user can simply use

```
plot(results)
```

to plot all time series contained in such a list. The dimensions and axes will automatically created to suit all series. Internally, the canvas is built using the most extreme values out of all time series. Subsequently further series are added using the basic *lines* command. Figure 6 shows a list of time series plotted using the *tlist* method for *plot*.

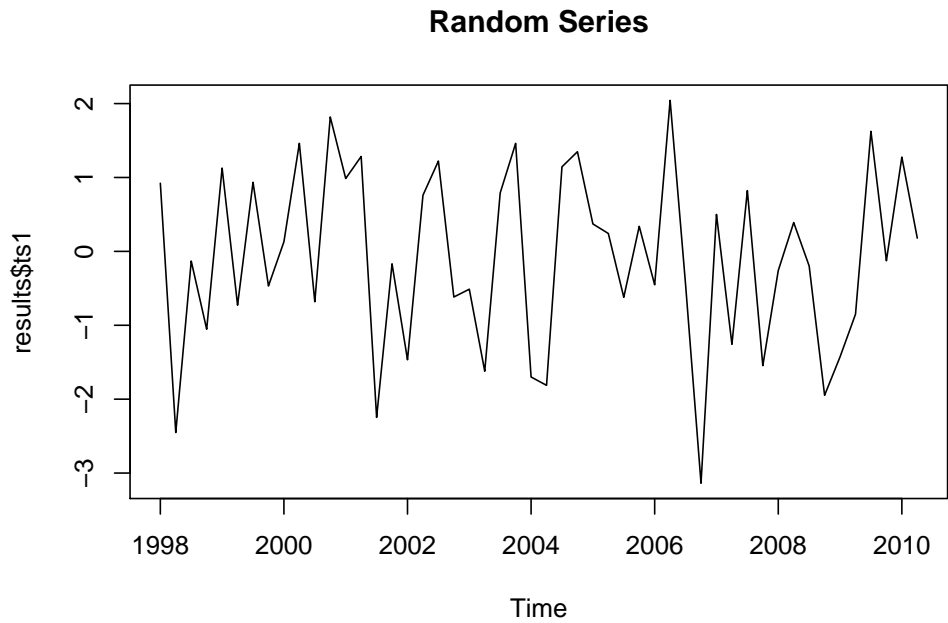


Figure 5: Plot with Dynamic Meta Information

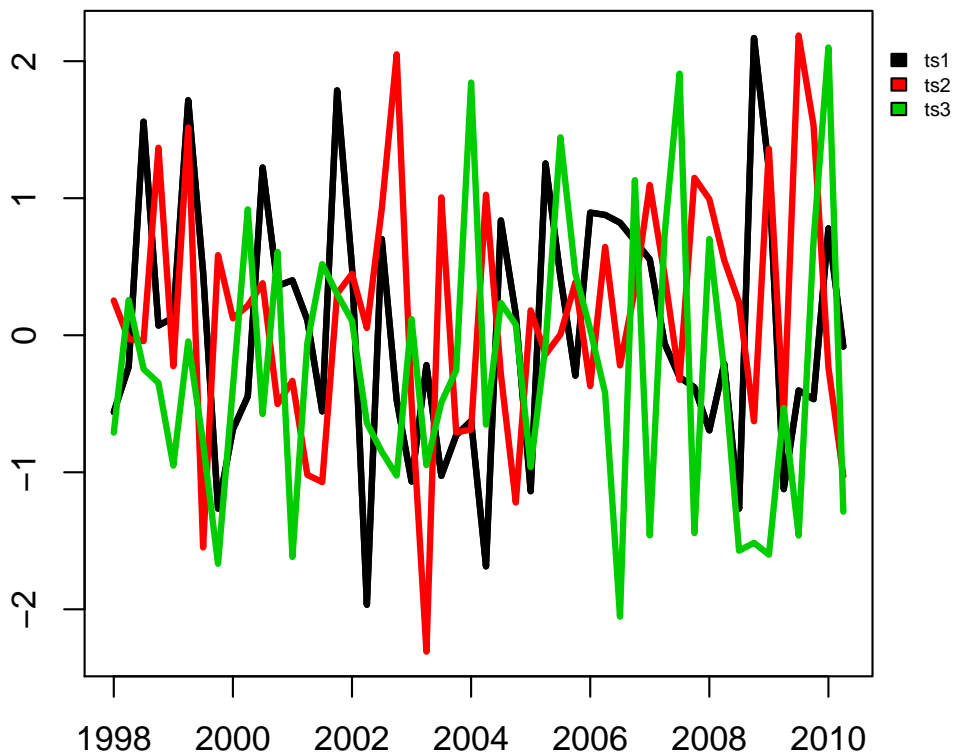


Figure 6: Plot a List of Time Series

5.5. Data Explorer

Another way to explore the content of a **timeseriesdb** based database is the *data explorer*. The data explorer is a simple web based graphical user interface (GUI) based on the popular **shiny** (Chang et al. 2015). Users can skim data interactively given a database connection. To start the GUI, run the code shown below, where *con* is a valid PostgreSQL connection object.

```
exploreDb(con)
```

Once the GUI has started up, the user has three options to query the database using the GUI: A *Key Based Query* lets the user look for matches with either the time series' primary key or within fixed meta information keys. The latter can be a relevant alternative when short cut aliases or legacy keys are used. The second option is to query localized meta information. Third, users can load pre-defined sets of time series that they had stored previously. In any case all query types return zero or more time series keys. Figure 7 shows the tab to build queries.

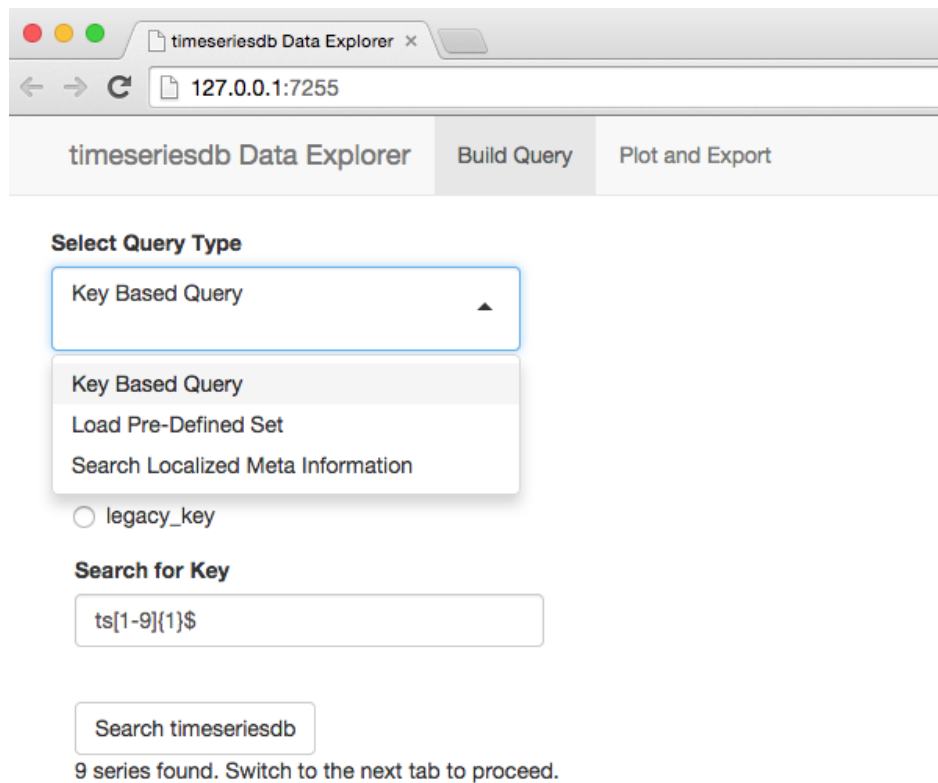


Figure 7: Web based Data Explorer: Build Queries

After creating a valid query, the user may switch to the *Plot and Export* tab and select those series that should be plotted, exported or saved in a set. After a selection is made, the keys of the selected series can be stored under a set name. By doing so, the set will be stored to the database and can be loaded again at a later stage. Also, time series can be exported as .csv files in either wide or long format. Also selected series will immediately plotted below to

give users an overview of the selected series. Note that, R's own localhost that hosts the **shiny** based data explorer and needs to be stopped if the user wants to go back to R's interactive console. Otherwise a separate R session needs to be started. Figure 8 shows the *Plot and Export* tab.

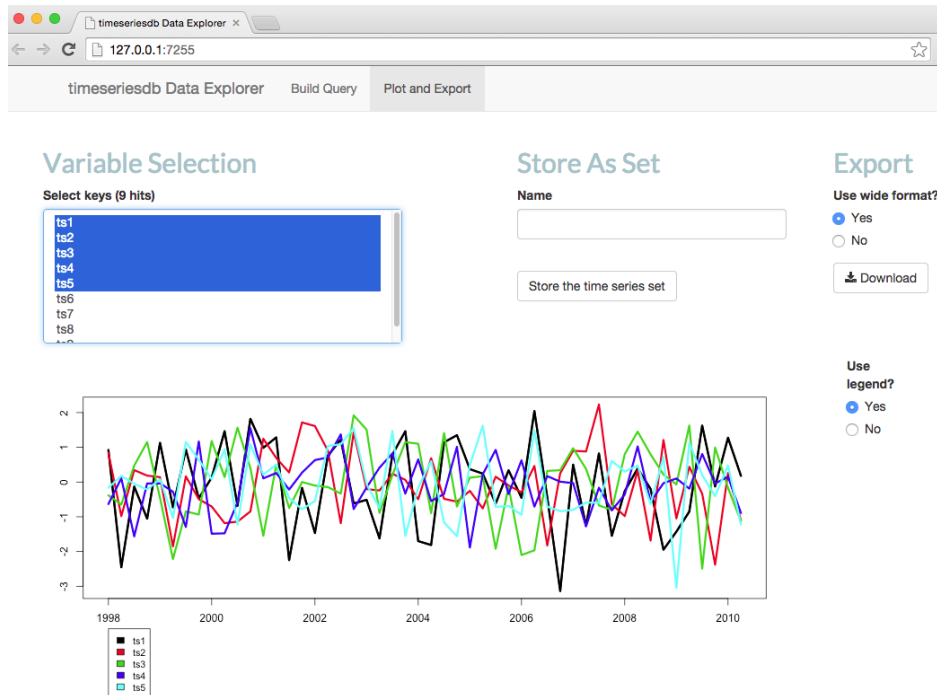


Figure 8: Web based Data Explorer: Plot and Export

6. Conclusion and Outlook

Though a brief paper cannot cover all details of **timeseriesdb**'s functionality, the general idea and core functionality has been presented in the previous chapters: **timeseriesdb** is a package to manage and archive time series data along with comprehensive meta information. Using R and PostgreSQL the closest existing CRAN package is **TSPostgreSQL**. Besides queries optimized for bulk storage and the use of the *hstore* data type, the main distinction of **timeseriesdb** from **TSPostgreSQL** is its support for extensive meta information. Using different storage types and optimizing queries comes at the cost of not being able to flexibly switch the DBMS like the **TSdbi** package family does (e.g. to MySQL), but PostgreSQL's advanced and comprehensive approach is worth the loss of this particular flexibility.

Designed to bridge the gap between domain specific computation and archiving data, **timeseriesdb** makes use of R's ability to handle numerous foreign data formats and to provide interfaces to many software packages including most standard relational DBMS. **timeseriesdb** aims at researchers who intend to use their own domain specific software package as opposed to implementing computation on the database level. Thus an important part of **timeseriesdb**'s future development strategy is to move SQL code that resides in R functions to the SQL level. Placing query code inside SQL functions will foster the development of interfaces to other

software packages but R. Though additional functionality such as plotting or merging time series will only be available in R using SQL functions will open up **timeseriesdb** to a broader audience whose software can connect to SQL databases. Further, upcoming extensions will continue to increase the support for a broader variety of frequencies including irregular time series. While this is basically possible with the current database structure already, future version will explore the use of **zoo** (Zeileis and Grothendieck 2005) and **xts** (Ryan and Ulrich 2013) as a general time series representation on the R level instead of R's basic *ts* class.

Thanks to advances in development of technologies that bring R to web servers, extending **timeseriesdb**'s functionality to create outputs that could be used to provide a web service seems appealing. Most prominently **shiny** which has already been used to create the web GUI inside **timeseriesdb**, along with shiny server could be used to create such a service. Also **opencpu** (Ooms 2014) provides a promising alternative approach that completely separates concerns and allows users to use functions of their own packages via a web service. In general future versions of **timeseriesdb** will extend the web interface of the package. Particular attention will be given to functionality such as shopping cart like data bookmarks in order to provide an infrastructure to host a data service that can be accessed from the web.

References

- Allaire, JJ, Jonathan McPherson, Yihui Xie, Hadley Wickham, Joe Cheng, and Jeff Allen. 2014. *Rmarkdown: Dynamic Documents for R*. <http://rmarkdown.rstudio.com>.
- Bannert, Matthias. 2015. *Timeseriesdb: Store and Manage Time Series in a Database*.
- Bengtsson, Henrik. 2015. *R.matlab: Read and Write MAT Files and Call MATLAB from Within R*. <http://CRAN.R-project.org/package=R.matlab>.
- Branchi, Mariagnese, Heinz Christian Dieden, Wim Haine, Csaba Horvath, Andrew Kanutin, and Linda Kezber. 2007. “Quarterly GDP Revisions in G-20 Countries: Evidence from the 2008 Financial Crisis.” *ECB Occasional Paper*, no. 74. <http://ssrn.com/abstract=1005928>.
- Chang, Winston, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. 2015. *Shiny: Web Application Framework for R*. <http://CRAN.R-project.org/package=shiny>.
- Conway, Joe, Dirk Eddelbuettel, Tomoaki Nishiyama, Sameer Kumar Prayaga, and Neil Tiffin. 2013. *RPostgreSQL: R Interface to the PostgreSQL Database System*. <http://CRAN.R-project.org/package=RPostgreSQL>.
- Databases, R Special Interest Group on. 2014. *DBI: R Database Interface*. <http://CRAN.R-project.org/package=DBI>.
- Denis Mukhin, David A. James, and Jake Luciani. 2014. *ROracle: OCI Based Oracle Database Interface for R*. <http://CRAN.R-project.org/package=ROracle>.
- Dowle, M, T Short, S Lianoglou, A Srinivasan with contributions from R Saporta, and E Antonyan. 2014. *Data.table: Extension of Data.frame*. <http://CRAN.R-project.org/package=data.table>.
- Gandrud, Christopher. 2013. *Reproducible Research with R and R Studio*. Boca Raton: Chapman; Hall/CRC.
- Gilbert, Paul. 2013a. *TSdbi: TSdbi (Time Series Database Interface)*. <http://CRAN.R-project.org/package=TSdbi>.
- . 2013b. *TSMYSQL: TSdbi Extensions for MySQL*. <http://CRAN.R-project.org/package=TSMYSQL>.
- . 2013c. *TSPostgreSQL: TSdbi Extensions for PostgreSQL*. <http://CRAN.R-project.org/package=TSPostgreSQL>.
- Koenker, Roger, and Achim Zeileis. 2009. “On Reproducible Econometric Research.” *Journal of Applied Econometrics* 24 (5). John Wiley & Sons, Ltd.: 833–47. doi:10.1002/jae.1083.
- Lang, Duncan Temple. 2014. *RJSONIO: Serialize R Objects to JSON, JavaScript Object Notation*. <http://CRAN.R-project.org/package=RJSONIO>.
- Leeper, Thomas J. 2014. “Archiving Reproducible Research with R and Dataverse.” *The R Journal* 6 (1): 151–58. <http://journal.r-project.org/archive/2014-1/leeper.pdf>.
- McCullough, B. D. 2009. “Open Access Economics Journals and the Market for Reproducible Economic Research.” In *Economic Analysis and Policy* 39.1, pp. 117–126.
- McCullough, B. D., and H. D. Vinod. 2003. “Verifying the Solution from a Nonlinear Solver: A Case Study.” *American Economic Review* 93 (3): 873–92. doi:10.1257/000282803322157133.
- Ooms, Jeroen. 2014. *Opencpu: The OpenCPU System for Embedded Scientific Computing and Reproducible Research*. <http://CRAN.R-project.org/package=opencpu>.

- R Core Team. 2015. *Foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, DBase*, . <http://CRAN.R-project.org/package=foreign>.
- Ryan, Jeffrey A., and Joshua M. Ulrich. 2013. *Xts: EXtensible Time Series*. <http://CRAN.R-project.org/package=xts>.
- Sax, Christoph. 2014. *Seasonal: R Interface to X-13ARIMA-SEATS*.
- Shrestha, Manik, and Marco Marini. 2013. “Quarterly GDP Revisions in G-20 Countries: Evidence from the 2008 Financial Crisis.” *IMF Working Paper*.
- Stokes, Houston H. 2004. “On the Advantage of Using Two or More Econometric Software Systems to Solve the Same Problem.” *Journal of Economic and Social Measurement* 29: 307–20.
- Wickham, Hadley, and Winston Chang. 2014. *Devtools: Tools to Make Developing R Code Easier*. <http://CRAN.R-project.org/package=devtools>.
- Xie, Yihui. 2013. *Dynamic Documents with R and Knitr*. Chapman; Hall/CRC. ISBN 978-1482203530.
- . 2014. “A Comprehensive Tool for Reproducible Research in R.” In *Implementing Reproducible Computational Research*., edited by Friedrich Leisch Victoria Stodden and Roger D. Peng. Chapman; Hall/CRC. ISBN 978-1466561595.
- . 2015. *Knitr: A General-Purpose Package for Dynamic Report Generation in R*. <http://CRAN.R-project.org/package=knitr>.
- Yang, Fangjin, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. *White Paper*: <http://static.druid.io/docs/druid.pdf>.
- Zeileis, Achim, and Gabor Grothendieck. 2005. “Zoo: S3 Infrastructure for Regular and Irregular Time Series.” *Journal of Statistical Software* 14 (6): 1–27. <http://www.jstatsoft.org/v14/i06/>.

7. Appendix

7.1. Installation Notes

R stable version

The stable version of the **timeseriesdb** R package itself can be downloaded and installed from CRAN (R's official repository). The package source as well as binaries for Windows and OS X are available from CRAN. The package can be installed following R's standard procedure to install packages either by running:

```
install.packages("timeseriesdb")
```

or using R's GUI.

R developer version

The developer version of **timeseriesdb** can be obtained from github.com/mbannert/timeseriesdb. The most convenient way to install the latest developer version from inside an R session is to use the **devtools** package (Wickham and Chang 2014):

```
library(devtools)
install_github('mbannert/timeseriesdb')
```

PostgreSQL

However, because **timeseriesdb** depends on **RPostgreSQL** to connect to PostgreSQL databases, the user needs to make sure that the PostgreSQL's own library and header files are present and can be found by **RPostgreSQL**. For Windows, this library called libpq is attached to the **RPostgreSQL** package and will thus be installed with the **RPostgreSQL** package. Hence Windows users should not experience not experience troubles.

For OS X and Linux the installation is a bit different when libpq is not present. For some Linux distributions the corresponding library can be obtained with the postgresql-devel package. Similarly on OS X, the user needs to make sure that libpq is present and can be found by **Rpostgresql**. It is recommended to use the **homebrew** package manager running `brew install postgresql`. OS X and Linux users should note that previously installed versions of PostgreSQL may not contain the libraries provided by postgresql-devel package.

Database setup

If you do not have a PostgreSQL database that contains a timeseries schema that suits timeseriesdb, create a schema called *timeseries* and run *setup.sql*. The file is located in *inst/sql* of your package folder. Start a psql client console from the *inst/sql* directory and run:

```
\i setup.sql
```

If a you are not familiar with running a PostgreSQL console, copy and paste the content of that file to the SQL window of your favorite GUI tool, e.g. PGAdmin and run it.

7.2. Benchmarks

Because **timeseriesdb** aims at storing a large number of time series, it is likely that **timeseriesdb** is used in bulk processes such reading thousands of time series into an R session, process them in R and write results back to the database. At this amount the speed of reading and writing to the database can become a substantial part of a process' runtime. Thus **timeseriesdb** strives to speed up reading and writing. The benchmark example shown in table 1 compares three different ways to read from the database.

	test	replications	elapsed	relative
1	<code>lapply(paste0("ts", 1:100), loopRead, con = con)</code>	100	79.38	15.41
2	<code>readAndSplit(paste0("ts", 1:100), con)</code>	100	21.51	4.18
3	<code>readTimeSeries(paste0("ts", 1:100), con)</code>	100	5.15	1.00

Table 1: Benchmark: Reading 100 Time Series, 100 replications

The first line shows benchmark results for a simple read function that reads a single time series from the database given its key. Using a function that reads a single series and loop over it seems to be the most intuitive solution for most users. However, doing so creates a new SELECT statement for every query that is executed. Plus, a new database transaction is started for every iteration. Particularly using multiple SELECT statements slows reading down considerably. Hence the approach shown in line 2 of the benchmark resolves the *hstore* data type and returns all observations from every time series to an R *data.frame*. Subsequently this *data.frame* is split by key and sorted by time using the **data.table** package (Dowle et al. 2014). The **data.table** package moves the split operation to C++ mainly and is therefore able to speed up the read process substantially, but splitting the data by key is still costly. The third line displayed in the benchmark is the version that is currently used within the **timeseriesdb** package. The function *readTimeSeries* makes use of a view that exposes entire time series records including key and frequency in a *JSON* object. Therefore the current read function only needs one SELECT statement and uses a WHERE IN clause. With the help of **RJSONIO**, the JSON objects can be resolved easily without using costly split actions. That makes the current version more than 15 times faster than the basic solution and still more than four time faster than having to split the set on R respectively C++ level.

Affiliation:

Matthias Bannert

KOF Swiss Economic Institute, ETH Zurich

Leonhardstrasse 21 LEE G 120 8045 Zurich Switzerland

E-mail: bannert@kof.ethz.ch

URL: <https://www.kof.ethz.ch/en/about-us/people/matthias-bannert/>