

Using **AdhereR** with various database technologies for processing very large datasets

Dan Dediu

21 November, 2018

Contents

Introduction	1
Prerequisites: load AdhereR and R markdown setup bits	2
Relational databases	2
Installing MySQL	3
Creating a sample database	3
Access the database and estimate adherence using explicit SQL	8
Connect to the database	8
How many patients are there?	8
How many events?	8
And how many events per patient?	9
Get the list of <code>patient_id</code> 's	11
Retreive the data for a given (set of) patient(s)	11
Compute CMA9 for all the patients and store it in the database	12
Plot a set of patients	16
Interactive plotting	18
Disconnect from the database	20
Using a non-local MySQL database	20
Optimisations and security	21
Use <code>dplyr</code> and <code>DBI</code> to transparently access the database	22
How about SAS and Stata	23
Finally, let's look at Hadoop and MapReduce !	24
Installing Hadoop 3.0.3 on Ubuntu 18.04	24
Install Java	24
Install Hadoop 3.0.3	24
Configure Hadoop	24
Test Hadoop	24
Installing RHadoop on Ubuntu 18.04	25
Set the needed environment variables	25
Install the needed R packages	25
Using Hadoop through RHadoop to compute CMA9	25
Load the libraries and initialize things	25
Store the <code>med.events</code> data in HDFS	26
Use MapReduce to compute CMA9	26
Load and use the results	27
Conclusions	29
References	30

Introduction

It is sometimes stated that **R** *cannot* be used with very large database as they don't fit in memory. Unfortunately, this has also been used as an argument against giving **AdhereR** a try, especially in contexts where it would be most useful, namely for processing (very) large datasets.

Here we try to dispel this worry by presenting various methods for dealing with such cases, focusing on data stored in “classic” *relational databases* (such as [MySQL](#), [MariaDB](#), [SQLite](#), [PostgreSQL](#), [Microsoft SQL Server](#) or [Oracle Database](#)) as well as on a widely-used paradigm for processing large datasets in a distributed manner, namely [Hadoop’s MapReduce](#) paradigm.

We will present several techniques that can be used to access such data from [AdhereR](#) (without attempting to load the whole of it in memory!), to compute the adherence to medication and/or generate plots, and to optionally store the results back into the database.

Please note that while the code here was tested on an [Ubuntu 18.04](#) “server” (an AMD Ryzen 7 2700X CPU with 8 physical cores and 16 logical ones, and 32 GB RAM) and a [macOS High Sierra](#) “client” (an early 2015 Macbook Air 11" 7,1 with an Intel Core i7-5650U CPU with 2 physical cores and 4 logical ones, and 8 GB RAM), actually running it (and, thus, compiling this very [Rmarkdown](#) script) requires a complex setup (detailed below). Therefore, we provide this vignette in compiled form as a [PDF](#) document¹ together with detailed instructions on how to install the required components for those users that want to reproduce or extend it.

Prerequisites: load [AdhereR](#) and [Rmarkdown](#) setup bits

Before we start, load [AdhereR](#) and various options concerning the figures:

```
library(AdhereR);  
  
# Various Rmarkdown output options:  
# center figures and reduce their file size:  
knitr::opts_chunk$set(fig.align = "center", dpi=150, dev="jpeg");
```

Relational databases

Relational databases have a venerable history and are very popular² in a multitude of settings, including those of potential interest for the computation and visualization of patterns of adherence to treatment. In such databases, data is organized in one or more *tables*, each table comprising several *columns* (or *variables*) and *rows* (or *entries*) – a representation familiar to most users of statistical software such as SPSS, SAS, Stata and R (in the latter, this is implemented by `data.frame` and friends). The querying of such databases is usually done through [SQL](#) (or *Structured Query Language*), which allows, among others, the selection of entries from such tables that meet certain requirements (for an introduction see, for example, Viescas & Hernandez (2008) among many others).

Usually, in practice, there *already* exists such a relational database that contains the relevant patient info organized in one or more tables, hosted by one of the many commercial or free/open source solutions (or *Relational Database Management Systems*, RDBMS) available, and which we can access using [SQL](#). For exemplification and reproducibility purposes, here we will also *create* these databases from the `med.events` dataset included in the [AdhereR](#) package (but these steps are obviously not part of the actual exploitation of the database).

We will focus here on [MySQL](#), but these can be applied to other RDBMS with minimal changes. [MySQL](#) is free and widely used, being a good example from the wider class of such systems. [MySQL](#) (and its close relative, [MariaDB](#)) is a stand-alone *server* that can be accessed (locally or remotely) by various *clients* who use [SQL](#) to manipulate the stored data. Thus, [MySQL](#) stands for the usual scenario where the patient, prescription and event data are stored in a centralized RDBMS (possibly hosted on a dedicated hardware and software

¹We provide only the [PDF](#) within the package itself as the [HTML](#) form is rather big due to the embedded images and generates a [NOTE](#) when building the package. Moreover, to void an unsuccessful attempt at compiling the vignette during the package build, the actual `.Rmd` source and related images are in the `specialVignettes` subfolder of the package.

²Despite the development of alternative architectures (generically known as “[NoSQL](#)”; see Harrison (2015)), RDBMSs are still extremely popular and will very probably continue to be so for the foreseeable future.

infrastructure) which can be queried (possibly over a network or even the Internet) by different specialized clients who perform particular tasks with (parts of) the data.

Currently, there are several ways of accessing a RDBMS from R, and we will focus here on two:

- a) using `SQL` directly, and
- b) transparently generating `SQL` queries through the `dplyr` front-end.

At the time of this writing (November 2018), the [MySQL Community Server](#) is a free version of the MySQL RDBMS which can be installed on several Operating Systems including Microsoft Windows, Apple's macOS and various flavors of Linux and BSD (for details, please see <https://dev.mysql.com/downloads/mysql/>). [MariaDB](#) is an open source RDBMS that started as a fork of MySQL (and is still very similar to it) and can as well be installed on a multitude of Operating Systems (for details, please see <https://mariadb.org/download/>).

Installing MySQL

Generic installation instructions can be found on the MySQL Community Server's [website](#), while step-by-step instructions geared towards R can be found, for example, in the [Introduction to MySQL with R](#) and [Connecting R to MySQL/MariaDB](#), among others. [How To Install MySQL on Ubuntu 18.04](#) is oriented specifically for Ubuntu 18.04 LTS (which we use on our test machine). In the following, we will assume that MySQL (Community Server) was successfully installed on the local machine.

In our case, we are on a machine running Ubuntu 18.04 LTS and we follow the procedure described in [How To Install MySQL on Ubuntu 18.04](#), with the difference that we create a user named `adherentuser` with password `AdhereR123!` using the command

```
CREATE USER 'adherentuser'@'localhost' IDENTIFIED BY 'AdhereR123!';
```

instead of the generic

```
CREATE USER 'sammy'@'localhost' IDENTIFIED BY 'password';
```

(and please make sure you do not forget to grant the new user the needed privileges:)

```
GRANT ALL PRIVILEGES ON *.* TO 'adherentuser'@'localhost' WITH GRANT OPTION;
```

Creating a sample database

Normally, this step is superfluous as the data should already be present in the RDBMS. Nevertheless, for illustration purposes, we transfer here the `med.events` example dataset that comes included with `AdhereR` into several tables in a MySQL database.

Use MySQL Workbench to connect to the database as user `adherentuser` using the password `AdhereR123!`. On our system, this means starting it either from the Desktop Environment's start menu or from the command prompt by typing:

```
mysql-workbench
```

after which, using the menu `Database -> Connect to Database...` we made a local connection using the desired user and credentials. Further, as described in [Introduction to MySQL with R](#), create a database:

```
CREATE DATABASE med_events;
```

which should become visible in the left-hand panel under "schemas" (see Figure).

Then, either using MySQL Workbench or SQL commands, create *four* tables within the `med_events` database:

- `event_patients` with two columns:
 - `id` of type `INT(11)`, primary key and non-null, and
 - `patient_id` of type `INT(11)` and non-null,

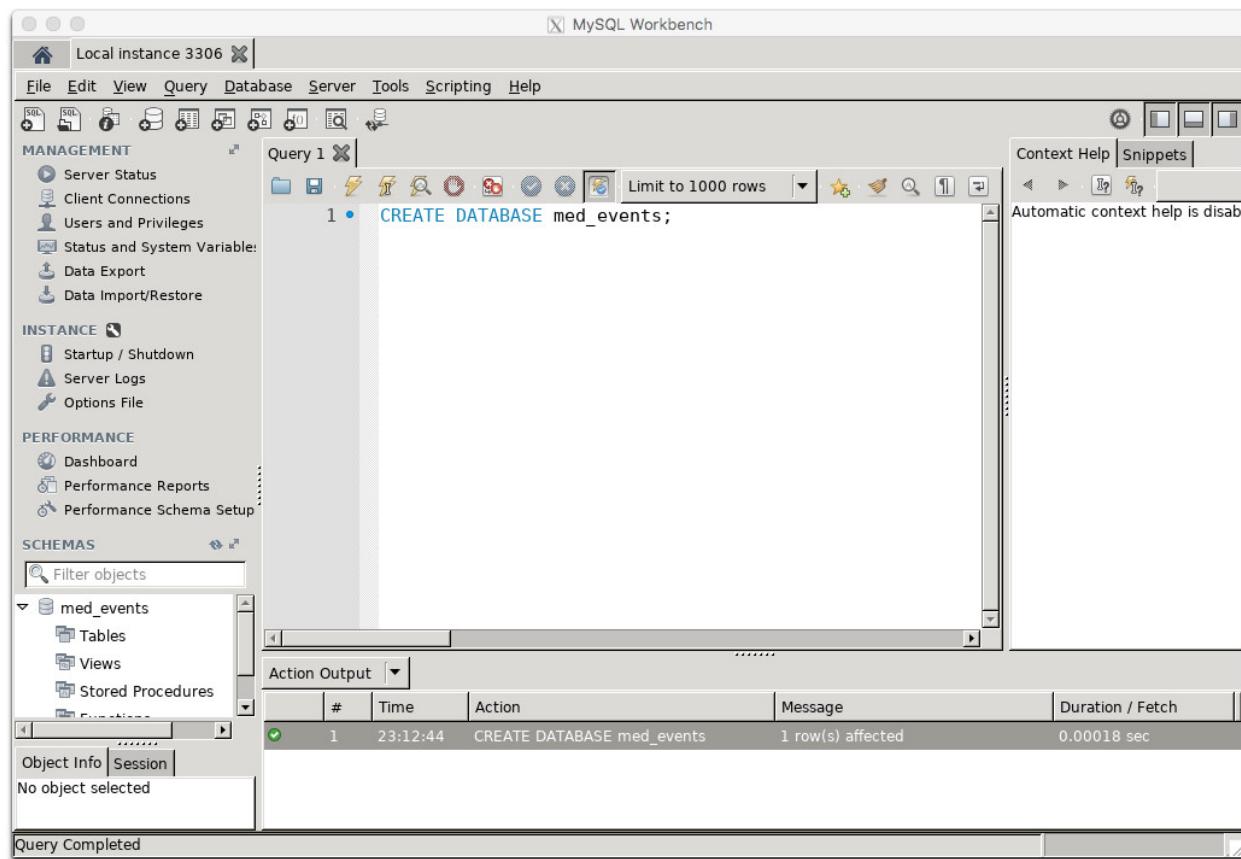


Figure 1: Creating a database using MySQL Workbench on Ubuntu 18.04 (screenshot taken on the macOS “client” using XQuartz for display).

- `event_date` with two columns:
 - `id` of type `INT(11)`, primary key and non-null, and
 - `date` of type `DATE` and non-null,
- `event_info` with four columns:
 - `id` of type `INT(11)`, primary key and non-null,
 - `perday` of type `INT(11)` and non-null,
 - `category` of type `VARCHAR(45)` and non-null, and
 - `duration` of type `INT(11)` and non-null.
- `patients` with two columns:
 - `id` of type `INT(11)`, primary key and non-null, and
 - `sex` of type `CHAR(1)` and non-null,

The SQL commands for these could be:

```
CREATE TABLE `med_events`.`event_patients` (
  `id` INT NOT NULL,
  `patient_id` INT NOT NULL,
  PRIMARY KEY (`id`));

CREATE TABLE `med_events`.`event_date` (
  `id` INT NOT NULL,
  `date` DATE NOT NULL,
  PRIMARY KEY (`id`));

CREATE TABLE `med_events`.`event_info` (
  `id` INT NOT NULL,
  `perday` INT NOT NULL,
  `category` VARCHAR(45) NOT NULL,
  `duration` INT NOT NULL,
  PRIMARY KEY (`id`));

CREATE TABLE `med_events`.`patients` (
  `id` INT NOT NULL,
  `sex` CHAR(1) NOT NULL,
  PRIMARY KEY (`id`));
```

Now, we will fill these tables using the data from `AdhereR`'s `med.events` dataset from R itself. First, make sure the the package `RMariaDB` is installed on your system (otherwise simply install it as usual, whether from RStudio or with `install.packages("RMariaDB")`) and load the library:

```
library(RMariaDB);
```

Then, connect to the database:

```
med_events_db <- dbConnect(RMariaDB::MariaDB(),
  user="adherentuser",          # the username
  password="AdhereR123!",       # and password (insecure but ok here)
  dbname='med_events',          # which database
  host='localhost'              # on which host (here, local machine)
);
```

and, just to check that things are OK, list the tables:

```
db_res_tables <- dbListTables(med_events_db); db_res_tables;
```

```
## [1] "cma9_estimates" "event_date"      "event_info"      "event_patients"
## [5] "patients"
```

Clear the tables (in case there's some junk info in there):

```
# Truncate all the tables:
for( i in db_res_tables ) dbExecute(med_events_db, paste0("TRUNCATE ",i,";"));
```

Now, generate an extra column containing the `event_id` unique event id (really, just the row number) and make sure the format of the dates fits SQL's DATE type:

```
d <- med.events; # make working a copy
d$event_id <- 1:nrow(d); # simply the index of the event in med.events
d$DATE <- as.character(as.Date(d$DATE, format="%m/%d/%Y"),
                      format="%Y-%m-%d"); # use the expected format for SQL's DATE
```

Fill in the `patients` table (and allocate some random sexes which we won't really use at all):

```
tmp <- unique(d[,c("PATIENT_ID")],drop=FALSE); # select the relevant info
tmp$sex <- sample(c("F", "M"), size=nrow(tmp), replace=TRUE);
names(tmp) <- c("id", "sex"); # match the column names
# For convenience, write the whole data.frame in one go:
dbWriteTable(med_events_db, name="patients",
             value=tmp, row.names=FALSE, append=TRUE);
```

then spread the event info across the remaining tables:

```
# event_date:
tmp <- d[,c("event_id", "DATE")]; # select the relevant info
names(tmp) <- c("id", "date"); # match the column names
# For convenience, write the whole data.frame in one go:
dbWriteTable(med_events_db, name="event_date",
             value=tmp, row.names=FALSE, append=TRUE);

# event_info:
tmp <- d[,c("event_id", "PERDAY", "CATEGORY", "DURATION")]; # select the relevant info
names(tmp) <- c("id", "perday", "category", "duration"); # match the column names
# For convenience, write the whole data.frame in one go:
dbWriteTable(med_events_db, name="event_info",
             value=tmp, row.names=FALSE, append=TRUE);

# event_patients:
tmp <- d[,c("event_id", "PATIENT_ID")]; # select the relevant info
names(tmp) <- c("id", "patient_id"); # match the column names
# For convenience, write the whole data.frame in one go:
dbWriteTable(med_events_db, name="event_patients",
             value=tmp, row.names=FALSE, append=TRUE);
```

Check that the data was correctly written either using MySQL Workbench or from R:

```
# Fetch the whole results as they are small enough for display:
knitr::kable(dbGetQuery(med_events_db, "SELECT * FROM patients LIMIT 5;"),
             align=c("l","l"),
             caption="First 5 rows of the `patients` table (please note
that the `sex` may differ from your results and between runs).");
```

Table 1: First 5 rows of the `patients` table (please note that the `sex` may differ from your results and between runs).

id	sex
1	F
2	F
3	F
4	M
5	M

```
knitr::kable(dbGetQuery(med_events_db, "SELECT * FROM event_date LIMIT 5;"),
  align=c("l","l"),
  caption="First 5 rows of the `event_date` table.");
```

Table 2: First 5 rows of the `event_date` table.

id	date
1	2033-04-26
2	2033-07-04
3	2033-08-03
4	2033-08-17
5	2033-10-13

```
knitr::kable(dbGetQuery(med_events_db, "SELECT * FROM event_info LIMIT 5;"),
  align=c("l","l"),
  caption="First 5 rows of the `event_info` table.");
```

Table 3: First 5 rows of the `event_info` table.

id	perday	category	duration
1	4	medA	50
2	4	medB	30
3	4	medB	30
4	4	medB	30
5	4	medB	30

```
knitr::kable(dbGetQuery(med_events_db, "SELECT * FROM event_patients LIMIT 5;"),
  align=c("l","l"),
  caption="First 5 rows of the `event_patients` table.");
```

Table 4: First 5 rows of the `event_patients` table.

id	patient_id
1	1
2	1
3	1
4	1
5	1

Finally, don't forget to close the connection to the database:

```
dbDisconnect(med_events_db);
```

Access the database and estimate adherence using explicit SQL

The first method uses explicit SQL statements directed at the MySQL server through the R library RMariaDB (despite its name, it also works for MySQL).

First, make sure it is installed on your system (for example by running the following or, if you're using RStudio, by checking the Packages panel):

```
require(RMariaDB); # there should be no warnings
```

If not, install it either using RStudio's *Tools -> Install packages* menu, or by running:

```
install.packages("RMariaDB", dep=TRUE); # also install the dependencies
```

(please make sure you check for any errors and follow any indication; for example, on Ubuntu 18.04 it is necessary to install prior to this the libmysqlclient-dev package using, for example, `apt install libmysqlclient-dev`).

Now, load it:

```
library(RMariaDB);
```

We will next illustrate a few scenarios concerning the computation and plotting of patterns of adherence accessing data from the database.

Connect to the database

```
med_events_db <- dbConnect(RMariaDB::MariaDB(),           # works also for MySQL
                            user="adherentuser",        # the username
                            password="AdhereR123!",     # and password (insecure but ok here)
                            dbname='med_events',        # which database
                            host='localhost'            # on which host (here, local machine)
);
db_res_tables <- dbListTables(med_events_db); db_res_tables; # check things are ok

## [1] "cma9_estimates" "event_date"      "event_info"      "event_patients"
## [5] "patients"
```

How many patients are there?

```
no_patients <- dbGetQuery(med_events_db,
                           # overkill, as `id` is the primary key:
                           "SELECT COUNT(DISTINCT id) FROM patients;");
no_patients <- as.numeric(no_patients[1,1]); # single value: convert it to numeric
no_patients;

## [1] 100
```

How many events?

```
no_events <- dbGetQuery(med_events_db,
                        "SELECT COUNT(DISTINCT id) FROM event_info;");
```

```

no_events <- as.numeric(no_events[1,1]); # single value: convert it to numeric
no_events;

## [1] 1080

```

And how many events per patient?

```

dbGetQuery(med_events_db,
  "SELECT patient_id, COUNT(*) FROM event_patients GROUP BY patient_id;");

```

	patient_id	COUNT(*)
## 1	1	24
## 2	2	8
## 3	3	28
## 4	4	6
## 5	5	8
## 6	6	9
## 7	7	6
## 8	8	14
## 9	9	9
## 10	10	14
## 11	11	6
## 12	12	7
## 13	13	5
## 14	14	24
## 15	15	13
## 16	16	22
## 17	17	10
## 18	18	6
## 19	19	6
## 20	20	11
## 21	21	10
## 22	22	7
## 23	23	11
## 24	24	9
## 25	25	11
## 26	26	12
## 27	27	13
## 28	28	10
## 29	29	14
## 30	30	11
## 31	31	3
## 32	32	6
## 33	33	4
## 34	34	11
## 35	35	11
## 36	36	11
## 37	37	8
## 38	38	9
## 39	39	10
## 40	40	11
## 41	41	6
## 42	42	7
## 43	43	20

## 44	44	7
## 45	45	11
## 46	46	18
## 47	47	10
## 48	48	12
## 49	49	6
## 50	50	9
## 51	51	6
## 52	52	7
## 53	53	4
## 54	54	5
## 55	55	5
## 56	56	11
## 57	57	4
## 58	58	16
## 59	59	18
## 60	60	7
## 61	61	10
## 62	62	8
## 63	63	6
## 64	64	11
## 65	65	19
## 66	66	20
## 67	67	14
## 68	68	8
## 69	69	6
## 70	70	10
## 71	71	11
## 72	72	6
## 73	73	10
## 74	74	8
## 75	75	13
## 76	76	11
## 77	77	6
## 78	78	25
## 79	79	23
## 80	80	19
## 81	81	7
## 82	82	10
## 83	83	9
## 84	84	12
## 85	85	9
## 86	86	13
## 87	87	10
## 88	88	14
## 89	89	6
## 90	90	12
## 91	91	7
## 92	92	18
## 93	93	7
## 94	94	11
## 95	95	11
## 96	96	22
## 97	97	12

```
## 98      98      8
## 99      99      5
## 100     100     16
```

Get the list of patient_id's

```
patient_ids <- dbGetQuery(med_events_db, "SELECT id FROM patients;"); # a data.frame
patient_ids <- patient_ids$id; # convert it to a vector
patient_ids;
```

```
## [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
## [35] 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
## [52] 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
## [69] 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
## [86] 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

Retreive the data for a given (set of) patient(s)

We don't assume here that we can load a lot of patients in memory at a time (after all, the whole point of this exercise is to keep the data in the database as much as possible), so we wrote a simple (but clearly not the most efficient!) function that retrieves the info for a given (set of) patient(s):

```
# Given a (set of) patient(s) ID(s) and a database connection,
# retreive his/her/their info and
# return it as a data.frame (or NULL)
retreive_patients_info <- function(patient_ids, # a single value or a vector
                                    db_connection) # the database connection
{
  s <- dbGetQuery(db_connection, paste0(" SELECT event_date.id,
                                         event_date.date,
                                         event_info.category,
                                         event_info.duration,
                                         event_info.perday,
                                         event_patients.patient_id,
                                         patients.sex
                                         FROM event_date
                                         JOIN event_info
                                         ON event_info.id = event_date.id
                                         JOIN event_patients
                                         ON event_patients.id = event_info.id
                                         JOIN patients
                                         ON patients.id = event_patients.patient_id
                                         WHERE patients.id IN (",
                                         paste0(patient_ids,collapse=","),
                                         "));"));

  if( nrow(s) < 1 )
  {
    return (NULL);
  } else
  {
    return (s);
  }
}
```

Compute CMA9 for all the patients and store it in the database

Also, for the sake of the exercise, we cannot assume that we can store the computed CMA values for all the patients in a vector in R, so, as soon as we estimate it for a single patient, we write it back to the database. For this, we will create a new table named `cma9_estimates` (with two columns: `patient_id` and `cma`) and we make sure it's empty:

```
dbExecute(med_events_db, "CREATE TABLE IF NOT EXISTS med_events.cma9_estimates (
    patient_id INT NOT NULL,
    cma REAL NULL,
    PRIMARY KEY (patient_id));" ); # create it if not already there

## [1] 0
dbExecute(med_events_db, "TRUNCATE cma9_estimates;" ); # make sure it's empty

## [1] 0
```

For each patient in turn, extract his/her data from the database, estimate CMA9 and write back this estimate to the database (also, for the sake of illustration, we will assume that we can't even store the whole list of patient ID's, so we ask for each individual patient by selecting individual rows in the `patients` table):

```
# Time the whole thing:
start.time <- Sys.time();

# First (re)get the number of patients in the whole database:
no_patients <- dbGetQuery(med_events_db, "SELECT COUNT(DISTINCT id) FROM patients;");
no_patients <- as.numeric(no_patients[1,1]); # single value: convert it to numeric
no_patients;

## [1] 100

# Select each patient in turn by its row number (remeber: rows count starts at 0 in SQL!):
for( i in 0:(no_patients-1) )
{
  # Get the patient ID in row i:
  patient_id <- dbGetQuery(med_events_db,
                            paste0("SELECT id FROM patients LIMIT ",i,",1;"));
  if( is.null(patient_id) || nrow(patient_id) < 1 )
  {
    warning(paste0("Cannot find patient number ",i+1,"!\n"));
    next;
  }
  patient_id <- as.numeric(patient_id[1,1]); # single value: convert it to numeric

  # Extract the patient's data:
  s <- retreive_patients_info(patient_id, med_events_db); # retreive all the data
  if( is.null(s) )
  {
    warning(paste0("Cannot retrieve data for patient '",patient_id,"'!\n"));
    cma <- NA;
  } else
  {
    # Estimate the CMA:
    cma <- CMA9(data=s, # compute the CMA
                  ID.colname="patient_id",
                  event.date.colname="date",
                  event.duration.colname="duration",
```

```

    event.daily.dose.colname="perday",
    medication.class.colname="category",
    followup.window.start=0,
    followup.window.start.unit="days",
    followup.window.duration=2*365,
    followup.window.duration.unit="days",
    observation.window.start=30,
    observation.window.start.unit="days",
    observation.window.duration=1*365,
    observation.window.duration.unit="days",
    date.format="%Y-%m-%d",
    parallel.backend="none");
if( is.null(cma) )
{
  warning(paste0("Cannot estimate CMA for patient '",patient_id,"'!\n"));
  cma <- NA;
} else
{
  cma <- getCMA(cma)$CMA[1]; # retrieve the estimate
}
}

# Write back this estimate (with replacement,
# if there's an already existing estimate for this patient):
result <- dbExecute(med_events_db,
                     paste0("REPLACE INTO cma9_estimates (patient_id, cma) VALUES (",
                           patient_id,
                           ", ",
                           ifelse(is.na(cma),"NULL",cma),")"));
}

end.time <- Sys.time();
cat(paste0("The SQL-mediated computation of CMA9 on the whole database took: ",
          round(difftime(end.time, start.time, units="secs"),2),
          " seconds\n"));

## The SQL-mediated computation of CMA9 on the whole database took: 2.17 seconds

```

Also, do it within R so we can compare the results of the two procedures:

```

# Also, time it using the same procedure:
start.time <- Sys.time();
cma9r <- CMA9(data=med.events,
                ID.colname="PATIENT_ID",
                event.date.colname="DATE",
                event.duration.colname="DURATION",
                event.daily.dose.colname="PERDAY",
                medication.class.colname="CATEGORY",
                followup.window.start=0,
                followup.window.start.unit="days",
                followup.window.duration=2*365,
                followup.window.duration.unit="days",
                observation.window.start=30,
                observation.window.start.unit="days",
                observation.window.duration=1*365,

```

```

    observation.window.duration.unit="days",
    date.format="%m/%d/%Y",
    parallel.backend="none");
end.time <- Sys.time();
cat(paste0("The computation of CMA9 in R on the whole database took: ",
          round(difftime(end.time, start.time, units="secs"),2),
          " seconds\n"));

## The computation of CMA9 in R on the whole database took: 0.3 seconds

# Retreive the results from the database:
cma9sql <- dbGetQuery(med_events_db, "SELECT * FROM cma9_estimates;");
knitr::kable(merge(getCMA(cma9r),
                   cma9sql,
                   by.x="PATIENT_ID", by.y="patient_id", all=TRUE),
            align=c("c","c","c"),
            col.names=c("Patient ID", "CMA9 (R)", "CMA9 (MySQL)"),
            caption="The estimation of CMA9 on all the patients comparing
            the `MySQL` and `R` estimates (they are, as expected, identical!).");

```

Table 5: The estimation of CMA9 on all the patients comparing the MySQL and R estimates (they are, as expected, **identical**!).

Patient ID	CMA9 (R)	CMA9 (MySQL)
1	0.9185229	0.9185229
2	0.7076512	0.7076512
3	0.9951294	0.9951294
4	0.2126893	0.2126893
5	0.3533546	0.3533546
6	0.2639304	0.2639304
7	0.6085971	0.6085971
8	0.9026890	0.9026890
9	0.6898670	0.6898670
10	0.7300316	0.7300316
11	0.1457626	0.1457626
12	0.4534314	0.4534314
13	0.2694568	0.2694568
14	0.6896727	0.6896727
15	0.4841183	0.4841183
16	1.0000000	1.0000000
17	0.4709437	0.4709437
18	0.2511233	0.2511233
19	0.2705104	0.2705104
20	0.3883306	0.3883306
21	0.4534097	0.4534097
22	0.3162253	0.3162253
23	0.6840264	0.6840264
24	0.2437649	0.2437649
25	0.4551645	0.4551645
26	0.6548924	0.6548924
27	0.6948941	0.6948941
28	0.7698630	0.7698630
29	0.8821918	0.8821918
30	0.9551252	0.9551252

Patient ID	CMA9 (R)	CMA9 (MySQL)
31	0.2641743	0.2641743
32	0.1103753	0.1103753
33	0.1433427	0.1433427
34	0.6662712	0.6662712
35	0.4545709	0.4545709
36	0.9002090	0.9002090
37	0.5391753	0.5391753
38	1.0000000	1.0000000
39	0.6711226	0.6711226
40	0.7778944	0.7778944
41	0.1394641	0.1394641
42	0.7290265	0.7290265
43	1.0000000	1.0000000
44	0.3985459	0.3985459
45	0.7625689	0.7625689
46	1.0000000	1.0000000
47	0.8575342	0.8575342
48	0.6893408	0.6893408
49	0.3937378	0.3937378
50	0.8754789	0.8754789
51	0.5521455	0.5521455
52	0.7750355	0.7750355
53	0.1265823	0.1265823
54	0.2817497	0.2817497
55	0.3767123	0.3767123
56	0.4032023	0.4032023
57	0.2303609	0.2303609
58	0.8305149	0.8305149
59	0.9686888	0.9686888
60	0.3604903	0.3604903
61	0.6970132	0.6970132
62	0.3578026	0.3578026
63	0.2187364	0.2187364
64	0.3924057	0.3924057
65	1.0000000	1.0000000
66	0.9863014	0.9863014
67	0.9995849	0.9995849
68	0.3835616	0.3835616
69	0.4374202	0.4374202
70	0.7716222	0.7716222
71	0.4278826	0.4278826
72	0.3920769	0.3920769
73	0.5807266	0.5807266
74	0.3664650	0.3664650
75	1.0000000	1.0000000
76	0.8434442	0.8434442
77	0.1163283	0.1163283
78	0.8636008	0.8636008
79	1.0000000	1.0000000
80	0.9643836	0.9643836
81	0.3249420	0.3249420
82	0.8810842	0.8810842

Patient ID	CMA9 (R)	CMA9 (MySQL)
83	0.4690076	0.4690076
84	0.5523369	0.5523369
85	0.3790906	0.3790906
86	0.6165855	0.6165855
87	0.8610212	0.8610212
88	0.3254529	0.3254529
89	0.2532995	0.2532995
90	0.4516032	0.4516032
91	0.1971892	0.1971892
92	0.6665592	0.6665592
93	0.5505812	0.5505812
94	0.9027050	0.9027050
95	0.5671233	0.5671233
96	0.9110578	0.9110578
97	0.5739005	0.5739005
98	0.6509310	0.6509310
99	0.3393666	0.3393666
100	0.9844865	0.9844865

Plot a set of patients

Here we generate some plots of patients from the database.

```
# Extract the data of the patient(s) to plot:
# (we relax the stringency and assume we can hold a the patient IDs in memory :):
s <- retreive_patients_info(patient_ids[1:5], med_events_db);
cma0 <- CMA0(data=s,
  ID.colname="patient_id",
  event.date.colname="date",
  event.duration.colname="duration",
  event.daily.dose.colname="perday",
  medication.class.colname="category",
  followup.window.start.unit="days",
  followup.window.duration=2*365,
  followup.window.duration.unit="days",
  observation.window.start=30,
  observation.window.start.unit="days",
  observation.window.duration=1*365,
  observation.window.duration.unit="days",
  date.format="%Y-%m-%d",
  parallel.backend="none");
plot(cma0);

# Extract the data of the patient(s) to plot:
# (we relax the stringency and assume we can hold a the patient IDs in memory :):
s <- retreive_patients_info(patient_ids[1:5], med_events_db);
cma9 <- CMA9(data=s,
  ID.colname="patient_id",
  event.date.colname="date",
  event.duration.colname="duration",
  event.daily.dose.colname="perday",
  medication.class.colname="category",
```

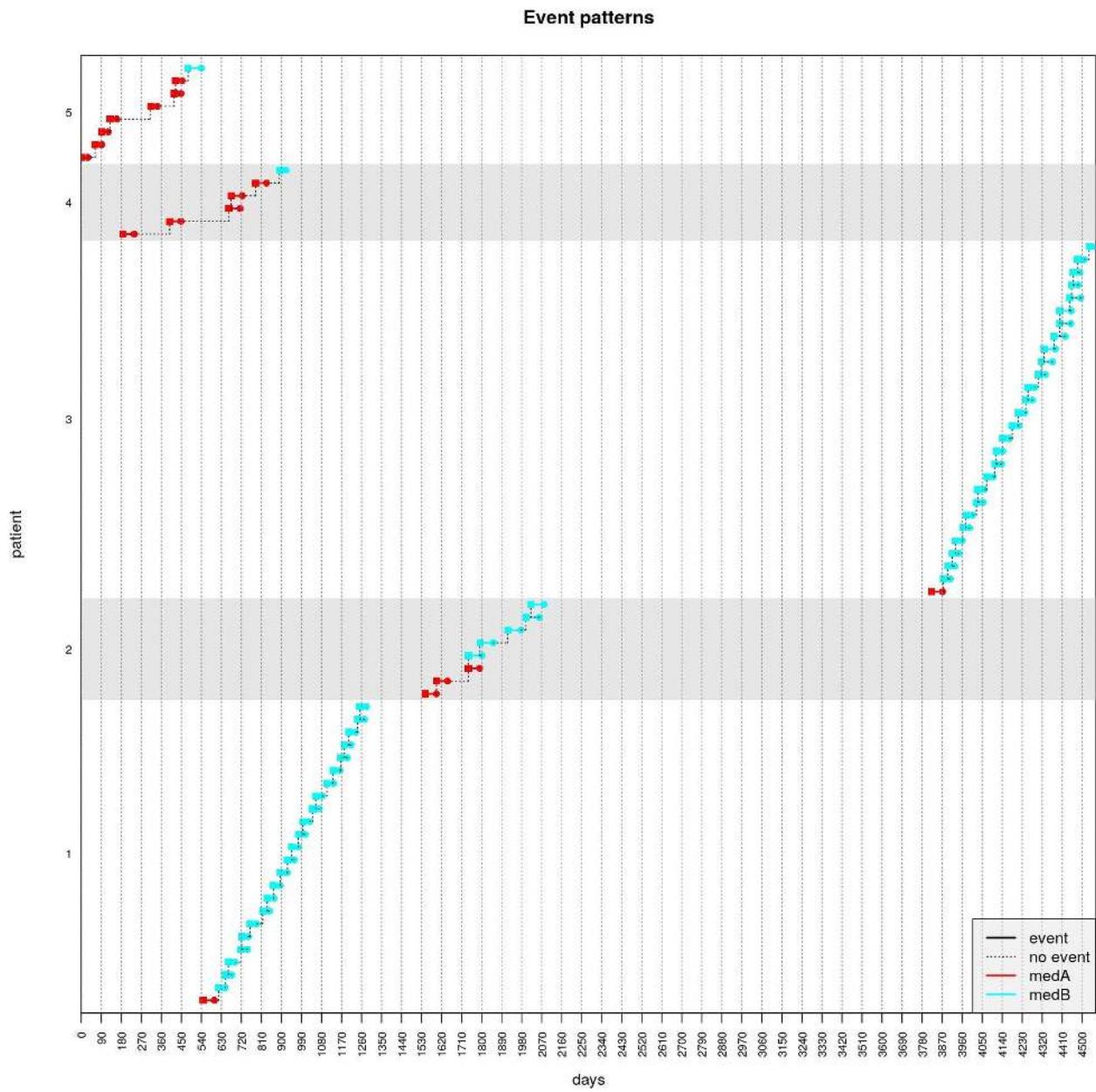


Figure 2: Plot of ‘CMA0‘ for the first 5 patients in the ‘MySQL‘ database using ‘SQL‘.

```

followup.window.start.unit="days",
followup.window.duration=2*365,
followup.window.duration.unit="days",
observation.window.start=30,
observation.window.start.unit="days",
observation.window.duration=1*365,
observation.window.duration.unit="days",
date.format="%Y-%m-%d",
parallel.backend="none");
plot(cma9);

```

Interactive plotting

For the interactive plotting of patients directly from the database (i.e., without local caching of data in R), we make use of the getter functions allowed by `plot_interactive_cma` through its function arguments `get.colnames.fnc`, `get.patients.fnc` and `get.data.for.patients.fnc`. As detailed in the help page for `plot_interactive_cma`, these arguments allow a user to use other types of data storage than the default `data.frame` by overriding the default behaviors for listing the data column names, listing all the patient IDs and getting the actual data for a (set of) patient ID(s), respectively.

Here, we redefined these functions as follows:

```

# Retreive the column names of the data for the first patient:
get.colnames.fnc.MySQL <- function(d)
{
  return(names(retrieve_patients_info(
    as.numeric(dbGetQuery(med_events_db,
      "SELECT id FROM patients LIMIT 0,1;")[1,1]),
    d)
  )));
}

# List all the patient IDs:
get.patients.fnc.MySQL <- function(d, idcol)
{
  s <- dbGetQuery(d, "SELECT id FROM patients;");
  return(s$id);
}

# Retreive the data for given (set of) patient ID(s):
get.data.for.patients.fnc.MySQL <- function(patientid, d, idcol)
{
  retrieve_patients_info(patientid, d);
}

```

resulting in the code (not run here as it needs an interactive R session with `AdhereR` and `Shiny`):

```

plot_interactive_cma(data=med_events_db, # use the MySQL connection as the data provider
  ID.colname="patient_id",
  event.date.colname="date",
  event.duration.colname="duration",
  event.daily.dose.colname="perday",
  medication.class.colname="category",
  date.format="%Y-%m-%d", # SQL's DATE specification
  backend=c("shiny","rstudio")[1], # use Shiny
  use.system.browser=TRUE, # use the system web browser
  # Override the getter functions to use the MySQL database

```

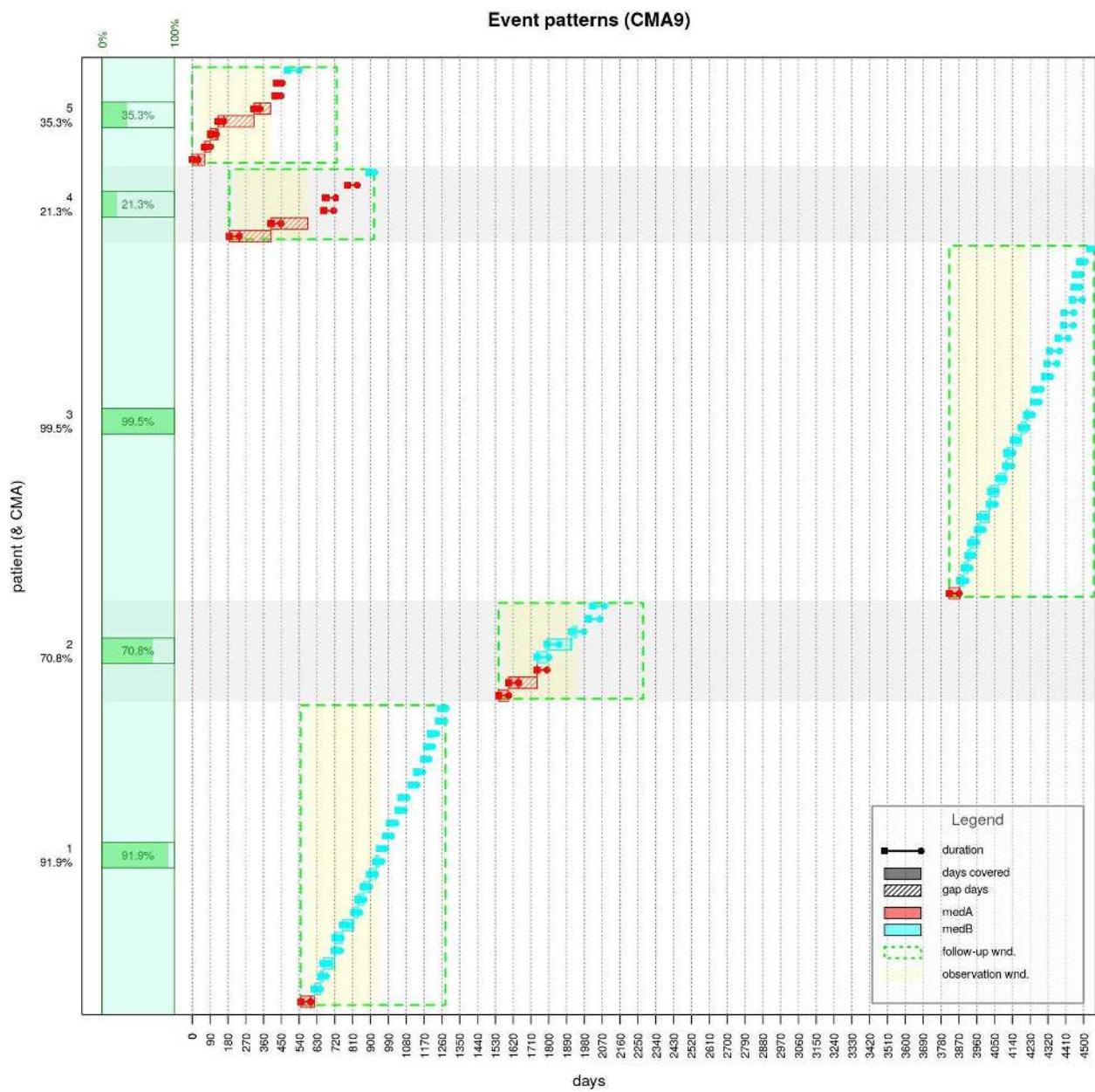


Figure 3: Plot of ‘CMA9’ for the first 5 patients in ‘MySQL’ database using ‘SQL’.

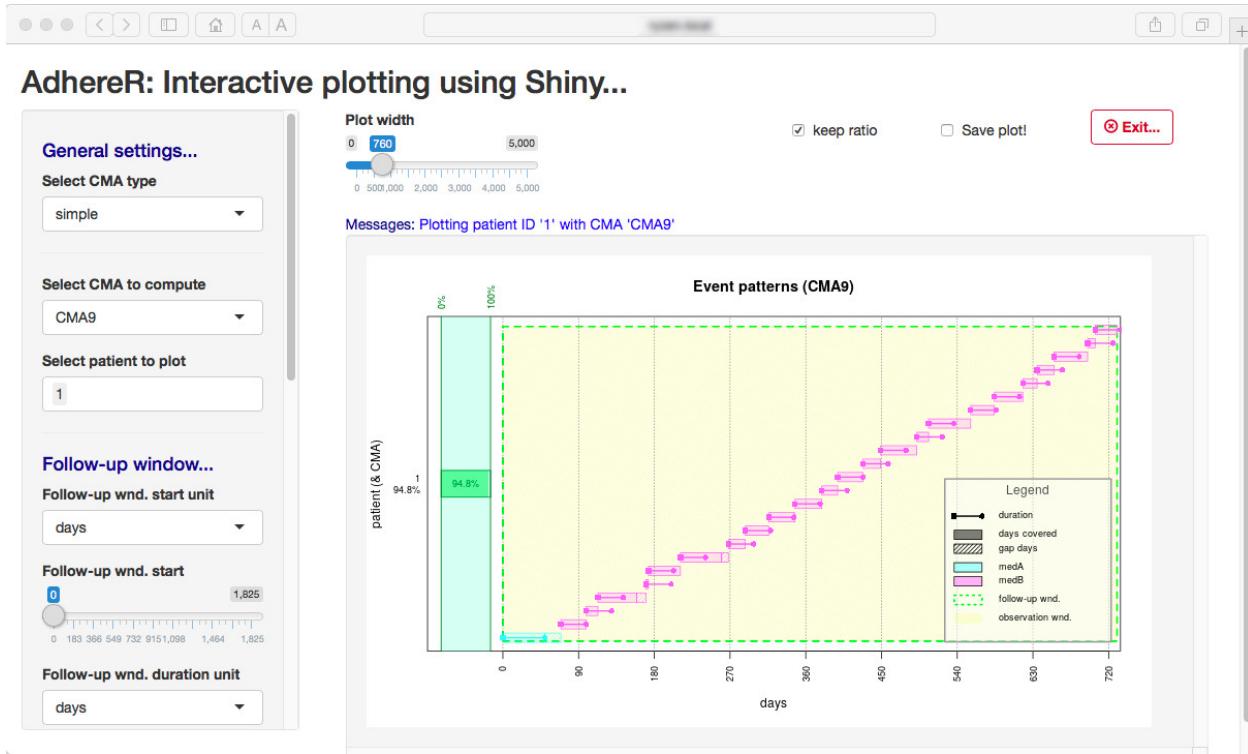


Figure 4: Screenshot of interactive plotting directly from a MySQL database using Shiny and the system browser (here, `Safari` on a `macOS High Sierra`).

```

# (please note that, for consistency, we must keep the
# function arguments even if we don't use all of them):
get.colnames.fnc=get.colnames.fnc.MySQL,
get.patients.fnc=get.patients.fnc.MySQL,
get.data.for.patients.fnc=get.data.for.patients.fnc.MySQL
);

```

Disconnect from the database

Finally, we need to disconnect from the database:

```
dbDisconnect(med_events_db);
```

Using a non-local MySQL database

Usually, the MySQL database is located on a *different* machine on a network (possibly even somewhere else on the Internet). We will show here how to access the same MySQL database we used so far (let's called it here the “*server*”) from a different machine (a Macbook Air 11" running `macOS High Sierra` with R 3.4.4; called the “*client*”).

First, make sure that the `RMariaDB` package is installed on the “*client*”.

Second, make sure that the user `adherentuser` is allowed to remotely access the MySQL database; for our setup, we did the following on the “*server*” (as suggested in <https://stackoverflow.com/questions/8380797/enable-remote-mysql-connection-error-1045-28000-access-denied-for-user> and <https://stackoverflow.com/>

[questions/8380797/enable-remote-mysql-connection-error-1045-28000-access-denied-for-user/21151255#21151255](https://stackoverflow.com/questions/8380797/enable-remote-mysql-connection-error-1045-28000-access-denied-for-user/21151255#21151255)):

- as `root` in MySQL (obtained by running, for example, `mysql -u root -p` in a terminal), execute:

```
GRANT ALL PRIVILEGES ON *.*  
TO 'adherentuser'@'%'  
IDENTIFIED BY 'AdhereR123!'  
WITH GRANT OPTION;  
FLUSH PRIVILEGES;
```

- also as `root`, edit the `/etc/mysql/my.cnf` (or the `/etc/mysql/mysql.conf.d/mysqld.cnf` on newer versions of MySQL) and replace the line `bind-address = 127.0.0.1` by `bind-address = 0.0.0.0`
- restart MySQL (for example, by `sudo service mysql restart`).

Now, on the “client” we should be able to connect to the remote “server”:

```
med_events_db <- dbConnect(RMariaDB::MariaDB(),      # works also for MySQL  
                           user="adherentuser",      # the username  
                           password="AdhereR123!",    # and password (insecure but ok here)  
                           dbname='med_events',      # which database  
                           host='remote.server.sql' # the host's name or IP address  
);
```

and check that everything is OK:

```
db_res_tables <- dbListTables(med_events_db); db_res_tables;
```

followed by all the other things you would do with a local database, including closing it at the end (as shown above). Of course, the access and data transfer times will be worse than for a local database, but this can be addressed by judiciously selecting the patients one needs to process and by processing them in chunks of more than one patient.

Optimisations and security

Especially for remote databases (but also for local ones), the access and data transfer are potentially very slow; moreover, `AdhereR` is heavily optimized for processing (possibly in parallel) multiple patients (using `data.table` and other techniques). Therefore, probably the following ideas may help speed up the processing:

- as far as possible, information extraction and preparation should be done server-side (i.e., using SQL) instead of client-side (i.e., transferring raw data and processing it in R);
- transfer locally only the data that is strictly needed for computing the adherence and/or plotting (e.g., if computing CMA1, it doesn’t make any sense to also transfer the column containing the daily dose) and only for the patients for which this needs to be done (i.e., perform an SQL pre-selection of the patients and events);
- to minimize server querying and transfer, and to maximally use the optimization built in `AdhereR`, split the whole set of patients to process into chunks that are relatively large (but not too large so as not to fit comfortably in the client’s RAM even when processed in parallel);
- collect the computation results in R as much as possible and only write them back to the database in large chunks.

In these examples we stored the server name, username and password in clear in the R code: while this OK for this demo, it is a **very bad idea** in a production environment! There are various techniques for mitigating this, ranging from storing these info in a separate file that is read at run-time, to asking the user to interactively give the username and password, to using the `keyring` package (<https://github.com/r-lib/keyring>) – see also <https://db.rstudio.com/best-practices/managing-credentials/> for a discussion. Please chose the most appropriate one for your use-case, but whatever you do, ***don’t store your credentials in the R script!!!***

Use `dplyr` and `DBI` to transparently access the database

This is based on the more extended discussion in <https://db.rstudio.com/dplyr/> . While using SQL for interacting with the database is very flexible and allows for fine-grained optimization, it requires a certain level of expertise and can be argued to create “chimeras” of R and SQL that might be more difficult to understand, debug and maintain.

The R packages `dbplyr` and `DBI` offer an alternative way, where the SQL is mostly “hidden” behind a familiar R code. Moreover, this allows to use the same code to access a variety of databases, such as MySQL, PostgreSQL and even Google’s BigQuery.

For example, we can connect and list the patients with (of course, you need to install first the needed packages, such as `dbplyr`, `dplyr` and `RMySQL`):

```
library(dplyr);

## 
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
## 
##     filter, lag

## The following objects are masked from 'package:base':
## 
##     intersect, setdiff, setequal, union

db_dplyer <- DBI::dbConnect(RMySQL::MySQL(),
                           user="adherentuser",      # the username
                           password="AdhereR123!",    # and password (insecure but ok here)
                           dbname='med_events'        # which database
);

DBI::dbListTables(db_dplyer); # list the tables in the database

## [1] "cma9_estimates" "event_date"      "event_info"      "event_patients"
## [5] "patients"

db_dplyer_patients <- tbl(db_dplyer, "patients"); # connect to the "patients" table
# db_dplyer_patients; # print it if you want to
```

Let’s get the data from patient with id 1, compute CMA9 and plot it:

```
# connect to the various event info tables as well:
db_dplyer_event_date <- tbl(db_dplyer, "event_date");
db_dplyer_event_info <- tbl(db_dplyer, "event_info");
db_dplyer_event_patients <- tbl(db_dplyer, "event_patients");

# Select and join the various pieces of info for patient with patient_id == 1
# (Please note that the data data is actually transferred from MySQL to R only
# at the end, when invoking collect()!!!)
s <- inner_join(inner_join(db_dplyer_event_info,
                           # filter the events for patient 1:
                           db_dplyer_event_patients %>% filter(patient_id == 1),
                           by="id"), # join with event_info
                db_dplyer_event_date, by="id") %>% collect();
cma9 <- CMA9(data=s, # compute CMA9 as usual
              ID.colname="patient_id",
              event.date.colname="date",
```

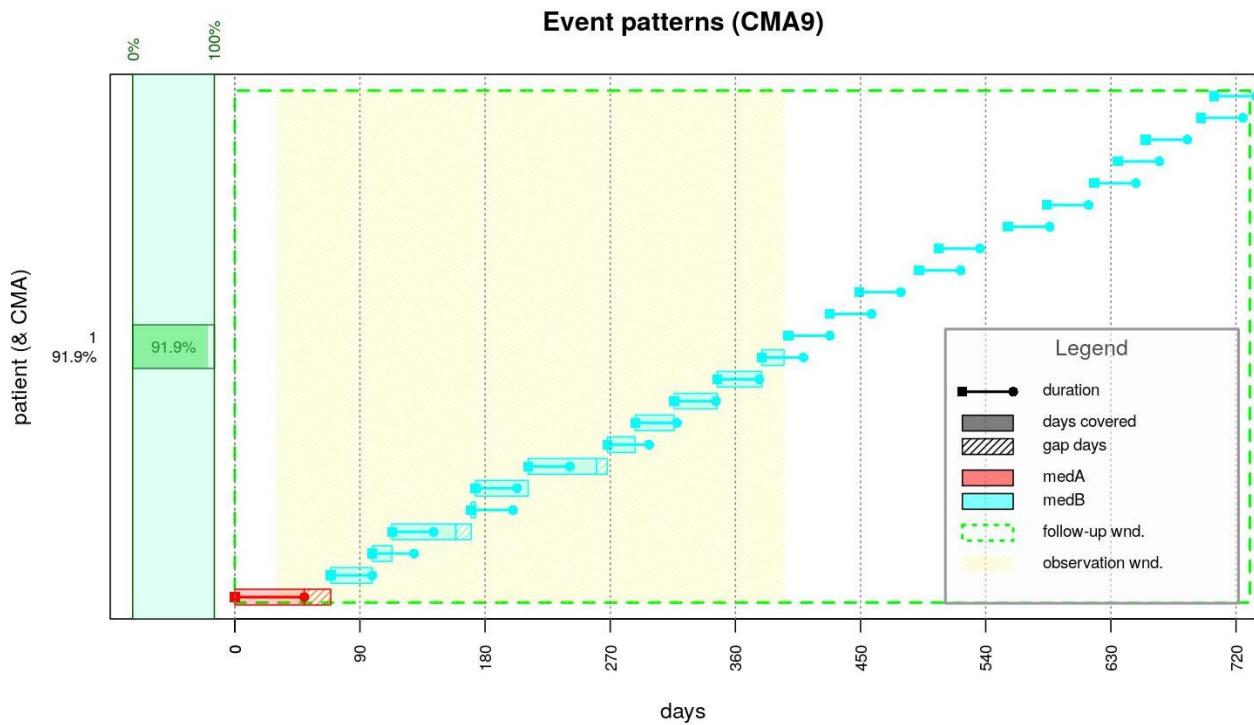


Figure 5: Plotting ‘CMA1‘ for patient with id ‘1‘ directly from a ‘MySQL‘ database using ‘dbplyr‘ and ‘DBI‘.

```

event.duration.colname="duration",
event.daily.dose.colname="perday",
medication.class.colname="category",
followup.window.start.unit="days",
followup.window.duration=2*365,
followup.window.duration.unit="days",
observation.window.start=30,
observation.window.start.unit="days",
observation.window.duration=1*365,
observation.window.duration.unit="days",
date.format="%Y-%m-%d",
parallel.backend="none");
plot(cma9); # plot it

```

Finally, close the connection to the database:

```
DBI::dbDisconnect(db_dplyer);
```

```
## [1] TRUE
```

How about SAS and Stata

If the dataset is relatively small, probably the easiest is to export it into a format that R can read (such as CSV); if this is not possible, then maybe one of the methods for importing their native formats in R may work (see, for example, <https://www.statmethods.net/input/importingdata.html> or https://cran.r-project.org/doc/manuals/r-devel/R-data.html#EpiInfo-Minitab-SAS-S_002dPLUS-SPSS-Stata-Systat).

However, if the dataset is (very) large, then probably one should either:

- export it into a RDBMS such as MySQL where R can access it (as described above), or
- experimentally, use ODBC to access it directly from SAS (which might or might not work).

Finally, let's look at Hadoop and MapReduce!

Apache Hadoop is a general framework for the distributed processing of large datasets using a simple programming paradigm (MapReduce; see, for example, <https://wiki.apache.org/hadoop/ProjectDescription>). Given that AdhereR can compute adherence estimates for sets of at least one patient at a time, it is very easy to embed within the MapReduce framework using a variety of approaches (see, among others, the outlines [here](#) or [here](#)).

Just as an example, we will install here Hadoop 3.0.3 on the Ubuntu 18.04 machine and use it to compute CMA9 on all the patients.

Installing Hadoop 3.0.3 on Ubuntu 18.04

We are following the tutorial [here](#). All of these happen in a `shell` terminal.

Install Java

```
sudo apt update
sudo apt install default-jdk
java -version
```

Install Hadoop 3.0.3

Download the 3.0.3 *binary* version:

```
cd ~/Downloads
wget http://mirrors.ircam.fr/pub/apache/hadoop/common/hadoop-3.0.3/hadoop-3.0.3.tar.gz
then unpack and move it to /usr/local/:
tar -xzvf hadoop-3.0.3.tar.gz
sudo mv hadoop-3.0.3 /usr/local/hadoop
```

Configure Hadoop

Add to `/usr/local/hadoop/etc/hadoop/hadoop-env.sh` the line `export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")`.

Test Hadoop

Start it first:

```
/usr/local/hadoop/bin/hadoop
```

and test it:

```
/usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-
mapreduce-examples-3.0.3.jar grep ~/input ~/grep_example 'allowed[.]*'
cat ~/grep_example/*
```

which should produce:

```
Output
19  allowed.
1  allowed
```

Installing RHadoop on Ubuntu 18.04

We are adapting here various guidelines for [RedHat](#) and [macOS](#). All these happen in R (unless specified).

Set the needed environment variables

```
Sys.setenv("HADOOP_HOME"="/usr/local/hadoop"); # Hadoop HOME
Sys.setenv("HADOOP_CMD"="/usr/local/hadoop/bin/hadoop"); # Hadoop binary
Sys.setenv("HADOOP_STREAMING"=  # Streaming
          "/usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.0.3.jar");
```

Install the needed R packages

(You probably already have some installed).

```
install.packages(c("rJava",
                   "Rcpp",
                   "RJSONIO",
                   "bitops",
                   "digest",
                   "functional",
                   "stringr",
                   "plyr",
                   "reshape2"),
                 dep=TRUE);
```

Now, manually download the `rhdfs`, `rbase` and `rnr2` packages from [Revolution Analytics' GitHub repository](#). In a shell terminal:

```
cd ~/Downloads
wget https://github.com/RevolutionAnalytics/rmr2/releases/download/3.3.1/rmr2_3.3.1.tar.gz
wget https://github.com/RevolutionAnalytics/rhdfs/blob/master/build/rhdfs_1.0.8.tar.gz?raw=true
wget https://github.com/RevolutionAnalytics/rbase/blob/master/build/rbase_1.2.1.tar.gz?raw=true
```

and then, in R:

```
install.packages(c("~/Downloads/HADOOP/rhdfs_1.0.8.tar.gz?raw=true",
                   "~/Downloads/HADOOP/rmr2_3.3.1.tar.gz",
                   "~/Downloads/HADOOP/rbase_1.2.1.tar.gz"),
                 repos=NULL, dep=TRUE);
```

Using Hadoop through RHadoop to compute CMA9

With all these in place, we can compute CMA9 for each participant, as follows (of course, this is not optimized in any sense). Everything here takes place in R.

Load the libraries and initialize things

```
library(rhdfs); # access to HDFS
```

```

## Loading required package: rJava
##
## HADOOP_CMD=/usr/local/hadoop/bin/hadoop
##
## Be sure to run hdfs.init()
hdfs.init(); # initialize HDFS

library(rmr2); # MapReduce in R

## Warning: S3 methods 'gorder.default', 'gorder.factor', 'gorder.data.frame',
## 'gorder.matrix', 'gorder.raw' were declared in NAMESPACE but not found
## Please review your hadoop settings. See help(hadoop.settings)

```

Store the `med.events` data in HDFS

First, we will store the `med.events` example `AdhereR` dataset in HDFS (the [Hadoop Distributed File System](#)) as a set of *key-value* pairs:

```

# Transfer med.events to HDFS as a set of key-value pairs:
med_events_hdfs <- to.dfs(keyval(med.events$PATIENT_ID, med.events));

```

This is *not* a very efficient and scalable way of doing it, but it works for this example (normally, the data would be transferred to HDFS using specialized tools such as [Sqoop](#)). What this does is create a handler (`med_events_hdfs`) to a temporary HDFS storage where the rows of the `med.events data.frame` (the *values*) are stored paired with the corresponding `PATIENT_ID`s (the *keys*). Importantly the `med_events_hdfs` handler ensures that the actual data is never fully loaded in memory, which is very important for large databases.

Use MapReduce to compute CMA9

```

s <- mapreduce(input=med_events_hdfs, # read the data from med_events_hdfs
               # the mapping function here doesn't do much;
               # k (= the key) = PATIENT_ID; v (= value) = med.events rows:
               map=function(k,v) keyval(k,v),
               # the reduce function computes CMA9 on the data (the values v)
               # associated with a given PATIENT_ID (the key k):
               reduce=function(k,v)
               {
                 cma <- CMA9(v,
                               ID.colname="PATIENT_ID",
                               event.date.colname="DATE",
                               event.duration.colname="DURATION",
                               event.daily.dose.colname="PERDAY",
                               medication.class.colname="CATEGORY",
                               followup.window.start=0,
                               followup.window.start.unit="days",
                               followup.window.duration=2*365,
                               followup.window.duration.unit="days",
                               observation.window.start=30,
                               observation.window.start.unit="days",
                               observation.window.duration=1*365,
                               observation.window.duration.unit="days",
                               date.format="%m/%d/%Y",
                               parallel.backend="none");

```

```

# Return the computed CMA as the value paired with the PATIENT_ID key:
keyval(k, getCMa(cma)[1, "CMA"]);
);

```

The basic idea is to first *map* the (key, value) pairs in the input (here, the PATEINT_IDs with their associated event info) to (potentially) different (key, value) pairs (here, we don't do basically anything at this stage), followed by *reducing* all the data associated with a key to a summary value (here, CMA9), returning a new summary (key, value) pair. The result `s` is also a pointer to the actual results stored in HDFS (temporarily), so that again we don't load anything large in memory.

Load and use the results

Just for the sake of illustration, only now do we load the full results in memory for display as a `data.frame` (to be compared with the other ways of computing CMA9):

```

# Only *now* force the loading of the full results in memory!
s <- from.dfs(s);

# Convert the (key, value) pairs to a nice data.frame format:
cma9hadoop <- data.frame("PATIENT_ID"=s$key, "CMA9"=s$val);

# And plot them for comparison with the other methods discussed here:
knitr::kable(merge(merge(getCMa(cma9r),
                         cma9sql,
                         by.x="PATIENT_ID", by.y="patient_id", all=TRUE),
                     cma9hadoop,
                     by="PATIENT_ID", all=TRUE),
               align=c("c", "c", "c", "c"),
               col.names=c("Patient ID", "CMA9 (R)", "CMA9 (MySQL)", "CMA9 (Hadoop)"),
               caption="The estimation of CMA9 on all the patients comparing the `R`, `MySQL` and `Hadoop` estimates (they are, as expected, identical!).");

```

Table 6: The estimation of CMA9 on all the patients comparing the R, MySQL and Hadoop estimates (they are, as expected, **identical!**).

Patient ID	CMA9 (R)	CMA9 (MySQL)	CMA9 (Hadoop)
1	0.9185229	0.9185229	0.9185229
2	0.7076512	0.7076512	0.7076512
3	0.9951294	0.9951294	0.9951294
4	0.2126893	0.2126893	0.2126893
5	0.3533546	0.3533546	0.3533546
6	0.2639304	0.2639304	0.2639304
7	0.6085971	0.6085971	0.6085971
8	0.9026890	0.9026890	0.9026890
9	0.6898670	0.6898670	0.6898670
10	0.7300316	0.7300316	0.7300316
11	0.1457626	0.1457626	0.1457626
12	0.4534314	0.4534314	0.4534314
13	0.2694568	0.2694568	0.2694568
14	0.6896727	0.6896727	0.6896727
15	0.4841183	0.4841183	0.4841183
16	1.0000000	1.0000000	1.0000000
17	0.4709437	0.4709437	0.4709437
18	0.2511233	0.2511233	0.2511233

Patient ID	CMA9 (R)	CMA9 (MySQL)	CMA9 (Haddop)
19	0.2705104	0.2705104	0.2705104
20	0.3883306	0.3883306	0.3883306
21	0.4534097	0.4534097	0.4534097
22	0.3162253	0.3162253	0.3162253
23	0.6840264	0.6840264	0.6840264
24	0.2437649	0.2437649	0.2437649
25	0.4551645	0.4551645	0.4551645
26	0.6548924	0.6548924	0.6548924
27	0.6948941	0.6948941	0.6948941
28	0.7698630	0.7698630	0.7698630
29	0.8821918	0.8821918	0.8821918
30	0.9551252	0.9551252	0.9551252
31	0.2641743	0.2641743	0.2641743
32	0.1103753	0.1103753	0.1103753
33	0.1433427	0.1433427	0.1433427
34	0.6662712	0.6662712	0.6662712
35	0.4545709	0.4545709	0.4545709
36	0.9002090	0.9002090	0.9002090
37	0.5391753	0.5391753	0.5391753
38	1.0000000	1.0000000	1.0000000
39	0.6711226	0.6711226	0.6711226
40	0.7778944	0.7778944	0.7778944
41	0.1394641	0.1394641	0.1394641
42	0.7290265	0.7290265	0.7290265
43	1.0000000	1.0000000	1.0000000
44	0.3985459	0.3985459	0.3985459
45	0.7625689	0.7625689	0.7625689
46	1.0000000	1.0000000	1.0000000
47	0.8575342	0.8575342	0.8575342
48	0.6893408	0.6893408	0.6893408
49	0.3937378	0.3937378	0.3937378
50	0.8754789	0.8754789	0.8754789
51	0.5521455	0.5521455	0.5521455
52	0.7750355	0.7750355	0.7750355
53	0.1265823	0.1265823	0.1265823
54	0.2817497	0.2817497	0.2817497
55	0.3767123	0.3767123	0.3767123
56	0.4032023	0.4032023	0.4032023
57	0.2303609	0.2303609	0.2303609
58	0.8305149	0.8305149	0.8305149
59	0.9686888	0.9686888	0.9686888
60	0.3604903	0.3604903	0.3604903
61	0.6970132	0.6970132	0.6970132
62	0.3578026	0.3578026	0.3578026
63	0.2187364	0.2187364	0.2187364
64	0.3924057	0.3924057	0.3924057
65	1.0000000	1.0000000	1.0000000
66	0.9863014	0.9863014	0.9863014
67	0.9995849	0.9995849	0.9995849
68	0.3835616	0.3835616	0.3835616
69	0.4374202	0.4374202	0.4374202
70	0.7716222	0.7716222	0.7716222

Patient ID	CMA9 (R)	CMA9 (MySQL)	CMA9 (Haddop)
71	0.4278826	0.4278826	0.4278826
72	0.3920769	0.3920769	0.3920769
73	0.5807266	0.5807266	0.5807266
74	0.3664650	0.3664650	0.3664650
75	1.0000000	1.0000000	1.0000000
76	0.8434442	0.8434442	0.8434442
77	0.1163283	0.1163283	0.1163283
78	0.8636008	0.8636008	0.8636008
79	1.0000000	1.0000000	1.0000000
80	0.9643836	0.9643836	0.9643836
81	0.3249420	0.3249420	0.3249420
82	0.8810842	0.8810842	0.8810842
83	0.4690076	0.4690076	0.4690076
84	0.5523369	0.5523369	0.5523369
85	0.3790906	0.3790906	0.3790906
86	0.6165855	0.6165855	0.6165855
87	0.8610212	0.8610212	0.8610212
88	0.3254529	0.3254529	0.3254529
89	0.2532995	0.2532995	0.2532995
90	0.4516032	0.4516032	0.4516032
91	0.1971892	0.1971892	0.1971892
92	0.6665592	0.6665592	0.6665592
93	0.5505812	0.5505812	0.5505812
94	0.9027050	0.9027050	0.9027050
95	0.5671233	0.5671233	0.5671233
96	0.9110578	0.9110578	0.9110578
97	0.5739005	0.5739005	0.5739005
98	0.6509310	0.6509310	0.6509310
99	0.3393666	0.3393666	0.3393666
100	0.9844865	0.9844865	0.9844865

Conclusions

We hope to have shown here that, far from being constrained by the “‘R’ can’t process large datasets that don’t fit in memory” myth, **AdhereR** can, in fact, nicely and easily interface with both “classical” relational databases (such as **MySQL**) and with newer technologies (such as Apache **Hadoop**), allowing it to access potentially huge databases (even across the Internet) and process them efficiently (even across massively parallel heterogeneous computational infrastructures).

Thus, **AdhereR** can seamlessly scale up from being used for the real-time visualization and computation of adherence of a small-to-medium local database (stored either in a “flat” CSV file or in an **SQLite** database) on a *consumer-grade laptop*, up to running on *large heterogenous computer clusters* where it can process huge amounts of data stored across several RDBMS tables (managed by, for example, **MySQL**) or as (key, value) pairs in a **Hadoop** **HDFS** file system. The examples (with actual code and step-by-step instructions) presented here should provide a starting point for real-world implementations optimized for the problems at hand.

There is a vast literature concerning both “classic” **SQL** and newer **NoSQL** databases, and there is an already mature ecosystem for accessing them from R. However, even for very specific cases where this is not feasible, we have provided a flexible framework for seamlessly using **AdhereR** from other programming languages (such as **Python**) for the computation and plotting of adherence.

References

Harrison, G. (2015). *Next Generation Databases: NoSQL and Big Data*. Apress.

Viescas, J. L., & Hernandez, M. J. (2008). *SQL Queries for Mere Mortals: A Hands-on Guide to Data Manipulation in SQL*. Addison-Wesley.