PL/R User's Guide - R Procedural Language

PL/R User's Guide - R Procedural Language

2. Installation

3. Functions and Arguments

4. Passing Data Values

5. Using Global Data

6. Database Access and Support Functions

6.1. Normal Support

6.2. RPostgreSQL Compatibility Support

7. PostgreSQL Support Functions

8. Aggregate Functions

9. Window Functions

10. Loading R Modules at Startup

11. R Function Names

12. Trigger Procedures

13. Inline Handler

14. Stored Procedures

15. Transactions in Stored Procedures

16. Custom Type (Tuple) Arguments in Window Functions

17. License

Overview

PL/R is a loadable procedural language that enables you to write PostgreSQL functions and triggers in the R programming language1. PL/R offers most (if not all) of the capabilities a function writer has in the R language. Commands are available to access the database via the PostgreSQL Server Programming Interface (SPI) and to raise messages via `elog()`. There is no way to access internals of the database backend. However the user is able to gain OS-level access under the permissions of the PostgreSQL user ID, as with a C function. Thus, any unprivileged database user should not be permitted to use this language. It must be installed as an untrusted procedural language so that only database superusers can create functions in it.

The writer of a PL/R function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. An implementation restriction is that PL/R procedures cannot be used to create input/output functions for new data types.

1. http://www.r-project.org/

Installation

All of the following presume that you have installed R before starting. From within R you can find R_HOME with `R.home(component="home")`

Redhat/Centos Family

This presumes you installed PostgreSQL using the PGDG repositories found here

```
yum install plr-nn
```

Where nn is the major version number such as 18 for PostgreSQL version 18.x

To set R_HOME for use by PostgreSQL.

First we need to customize the systemd service

```
systemctl edit postgresql-nn.service
```

again where nn is the major version of PostgreSQL installed on the system

Add the following to this file

```
[Service]
Environment=R_HOME=<The location of R_HOME found using R.home(component="home")>
```

Now restart PostgreSQL using

```
systemctl restart postgresql-nn
```

Debian deriviatives

This presumes you installed PostgreSQL using the PGDG repositories found here

```
apt-get install postgresql-nn-plr
```

In the `/etc/postgresql/nn/main` directory there is a file named environment. Edit this file and add the following:

```
R_HOME=<The location of R_HOME found using R.home(component="home")>
```

Compiling from source

If you are going to compile R from the source, then do the following:

```
./configure --enable-R-shlib --prefix=/opt/postgres_plr && make && make install
```

If you are going to compile PostgreSQL from the source, use the following commands from the untared and unzipped file downloaded from http://www.postgresql.org/ftp/source/:

Place source tar file in the contrib dir in the PostgreSQL source tree and untar it. The shared object for the R call handler is built and installed in the PostgreSQL library directory via the following commands (starting from /path/to/postgresql_source/contrib):

```
cd plr
make
make install
```

You may explicitly include the path of pg_config to `PATH`, such as

```
cd plr
PATH=/usr/pgsql-18/bin/:$PATH; USE_PGXS=1 make
echo "PATH=/usr/pgsql-18/bin/:$PATH; USE_PGXS=1 make install" | sudo sh
```

If you want to use git to pull the repository, run the following command before the make command:

```
git clone https://github.com/postgres-plr/plr
```

As of PostgreSQL 8.0.0, PL/R can also be built without the PostgreSQL source tree. Untar PL/R where ever you prefer. The shared object for the R call handler is built and installed in the PostgreSQL library directory via the following commands (starting from/path/to/plr):

```
cd plr
USE_PGXS=1 make
USE_PGXS=1 make install
```

Using https://www.msys2.org/ UCRT64 or MINGW32:

```
export R_HOME=/c/progra~1/R/R-4.6.1
export PATH=$PATH:/c/progra~1/PostgreSQL/18/bin
USE_PGXS=1 make
USE_PGXS=1 make install
```

Using https://www.msys2.org/ UCRT64 or MINGW32:

If R is built and installed using a sub-architecture, as explained in the section Sub-architectures in https://cran.r-project.org/doc/manuals/r-release/R-admin.html for example, in an R

```
R-x.y.z for Windows (32/64 bit) and version 4.1.3 or earlier
R-x.y.z for Windows (64 bit) and version 4.2.0 or later
```

that has been downloaded (and installed) from https://cran.r-project.org/bin/windows/base/old/

then, include the environment variable R_ARCH. For example R_ARCH=/x64 (or R_ARCH=/i386 as appropriate):

```
export R_HOME=/c/progra~1/R/R-4.6.1
export PATH=$PATH:/c/progra~1/PostgreSQL/18/bin
export R_ARCH=/x64
USE_PGXS=1 make
USE_PGXS=1 make install
```

```
export R_HOME=/c/progra~1/R/R-4.1.3
export PATH=$PATH:/c/progra~1/PostgreSQL/10/bin
export R_ARCH=/i386
USE_PGXS=1 make
USE_PGXS=1 make install
```

Note, R 4.2.0 and greater is not "single architecture." It is still "subarchitecture" with only 64bit. 32bit has been removed.

Compiling from source and using R for Windows 4.3.0 and later

PL/R that uses R for Windows 4.3.0 and later can no longer be compiled using Microsoft Visual Studio. One may read the following.

Status: CLOSED WONTFIX Bug 18544 - private_data_c Visual Studio 2022 R-4.3.0 Complex.h(81,21): syntax error: missing ';' before identifier 'private_data_c'

The new definition does not work with MSVC compilers because they don't support the C99 _Complex type

C Complex Numbers in C++?

Instead, for PL/R that uses R for Windows 4.3.0 and later, compile PL/R with https://www.msys2.org/ UCRT64.

Compiling from source using the meson build system

Needed is the PostgreSQL version 16 or later source code, libR installed, PATH set, and R_HOME set. One passes -DR_HOME=value to the `meson setup` command.

Alternately, needed are the PostgreSQL pre-compiled binaries. PostgreSQL can be a version lower than 16. Also needed are the libR installed, libpq installed, libpostgres configured and installed, PATH set, and R_HOME set. One passes -DR_HOME=value and -DPG_HOME=value2 to the `meson setup` command.

Installing from a Pre-Built "plr"

Win32 - adjust paths according to your own setup, and be sure to restart the PostgreSQL service after changing:

In Windows environment (generally):

```
R_HOME=C:\\Progra~1\\R\\R-4.6.1
Path=%PATH%;%R_HOME%\\x64\\bin
```

Detailed Windows Environment If wanting to install R 4.2.0 or later on a system older than Windows 10, then the following applies.

In R 4.2.0 or greater, support for 32-bit builds has been dropped.

R 4.2.0 and later uses UTF-8 as the native encoding on recent Windows systems (at least Windows 10 version 1903, Windows Server 2022 or Windows Server 1903). As a part of this change, R 4.2.0 and later uses UCRT as the C runtime. UCRT should be installed manually on systems older than Windows 10 or Windows Server 2016 before installing R.

This is documented at `CHANGES IN R 4.2.0` https://cran.r-project.org/doc/manuals/r-release/NEWS.html

Acquire UCRT64 through `Windows Update` or at the following URL query result: https://www.google.com/search?q=download+UCRT64

In a Windows environment, with a PL/R compiled using UCRT64 or MINGW32 or Microsoft Visual Studio https://github.com/postgres-plr/plr/releases/latest, with a PostgreSQL compiled with Microsoft Visual Studio https://www.enterprisedb.com/downloads/postgres-postgresql-downloads, and an R acquired from CRAN https://cran.r-project.org/bin/windows/base/ do the following.

First: Download and install PostgreSQL compiled with Microsoft Visual Studio https://www.enterprisedb.com/downloads/postgres-postgresql-downloads For R versions earlier than 4.3.0 Download PL/R compiled using Microsoft Visual Studio. For R versions greather or equal to 4.3.0 Download PL/R compiled using UCRT64 https://github.com/postgres-plr/plr/releases/latest

Unzip the plr.zip file into a folder, that is called the "unzipped folder". If your installation of PostgreSQL had been installed into "C:\Program Files\PostgreSQL\16", then from the unzipped PL/R folder, place the following

- .sql files and the plr.control file, all found in the "share\extension" folder into "C:\Program Files\PostgreSQL\16\share\extension" folder.
- plr.dll file found in the "lib" folder into "C:\Program Files\PostgreSQL\16\lib" folder.

Second: Install R with the feature checked [x] "Save version number in registry"." See the "Tip" item below.

Second Alternately: Acquire R from the same location and choose [ ] "Save version number in registry". At a Command Prompt run (and may have to be an Administrator Command Prompt) and using wherever your path to R may be, do:

```
setx R_HOME "C:\\Program Files\\R\\R-4.6.1" /M
```

Second Optionally: Acquire R from the same location and choose [ ] "Save version number in registry". Choose Control Panel -> System -> advanced system settings -> Environment Variables button. In the "System variables" area, create the System Variable, called R_HOME. Give R_HOME the value of the PATH to the R home, for example (without quotes) "C:\Program Files\R\R-4.6.1".

If you forgot to set the R_HOME environment variable (by any method), then (eventually) you may get this error:

```
postgres=# CREATE EXTENSION plr;
CREATE EXTENSION
postgres=# SELECT r_version();
ERROR:  environment variable R_HOME not defined
HINT:  R_HOME must be defined in the environment of the user that starts the postmaster process.
```

Third: Put the R.dll in your PATH. This is required, so do the following: Control Panel -> System -> Advanced System Settings -> Environment Variables button In the "System variables" area, choose the System Variable, called "Path". Click on the Edit button. Add the R.dll folder to the "Path". For example (without quotes), add "C:\Program Files\R\R-4.6.1\bin\x64" or or "C:\Program Files\R\R-4.1.3\bin\i386". If you are running R version 2.11 or earlier on Windows, the R.dll folder is different; instead of "bin\i386" or "bin\x64", it is "bin". Note, a 64bit compiled PL/R can only run with a 64bit compiled PostgreSQL. A 32bit compiled PL/R can only run with a 32bit compiled PostgreSQL. The last 32bit PostgreSQL was version ten(10) from https://www.enterprisedb.com/downloads/postgres-postgresql-downloads. Of course, you, yourself may try to compile a 32bit PostgreSQL using Microsoft Visual Studio.

Note, R 4.2.0 and greater is not "single architecture." It is still "subarchitecture" with only 64bit. 32bit has been removed.

Fourth: Restart the PostgreSQL cluster, do:

At a Command Prompt run (and you may have to be in an Administrator Command Prompt): Use the service name of whatever service your PostgreSQL is running under.

```
net stop postgresql-x64-18
```

Alternately, do the following: Control Panel -> Administrative Tools -> Services Find postgresql-x64-18 (or whatever service your PostgreSQL is running under). Right click and choose "Stop"

At a Command Prompt run (and you may have to be in an Administrator Command Prompt): Use the service name of whatever service your PostgreSQL is running under.

```
net start postgresql-x64-18
```

Alternately, do the following: Control Panel -> Administrative Tools -> Services Find postgresql-x64-18 (or whatever service your PostgreSQL is running under). Right click and choose "Start"

Tip R headers are required. Download and install R prior to building PL/R. R must have been built with the `--enable-R-shlib` option when it was configured, in order for the libR shared object library to be available.

Tip: Additionally, libR must be findable by your runtime linker. On Linux, this involves adding an entry in /etc/ld.so.conf for the location of libR (typically $R_HOME/bin or $R_HOME/lib), and then running ldconfig. Refer to `man ldconfig` or its equivalent for your system.

Tip: R_HOME must be defined in the environment of the user under which PostgreSQL is started, before the postmaster is started. Otherwise PL/R will refuse to load. See plr_environ(), which allows examination of the environment available to the PostgreSQL postmaster process.

Tip: On the Win32 platform, from a PL/R compiled by Microsoft Visual Studio, and from an R, installabled by an installer from https://cran.r-project.org/bin/windows/base/, R will consider a registry entry created by the R installer if it fails to find R_HOME environment variable. If you choose the installer option 'Save version number in registry', as explained in 'Does R use the Registry?' at https://cran.r-project.org/bin/windows/base/rw-FAQ.html there is no need to set R_HOME on this platform. Be careful removing older version of R as it may take away InstallPath entry away from HKLM/SOFTWARE/R-core/R a.k.a. Computer\HKEY_LOCAL_MACHINE\SOFTWARE\R-core\R.

General Windows ARM Environment

Experimental PL/R for Windows ARM is Now Available Some information is here (in chronological order):

Will R Work on 64-bit ARM Windows? - 2023/08/23 - Tomas Kalibera https://blog.r-project.org/2023/08/23/will-r-work-on-64-bit-arm-windows/

R on 64-bit ARM Windows - 2024/04/23 - Tomas Kalibera https://blog.r-project.org/2024/04/23/r-on-64-bit-arm-windows/index.html

Windows ARM64 comes to R-universe – Thursday, August 6, 2026 – By Jeroen Ooms https://ropensci.org/blog/2026/08/06/r-universe-winarm/

The R architecture is reverted to simplification:

- The architecture is only 64bit and is "single architecture." like the old Windows 95. Therefore R_ARCH and is not used

The installation instructions are the same as installation instructions for "PL/R for Windows" as described this markdown file with the following exceptions:

1. Experimental "PL/R for Windows ARM" can only run on a "Windows ARM" Operating system.
2. "R for Windows ARM" is available here: https://github.com/r-devel/windows-arm64/releases . Or the original location here https://github.com/r-devel/r-svn/actions; choose the "Build" action. Choose the artifact `Win-installer-aarch64`.
3. EDB DOES NOT build a PostgreSQL for Windows ARM. https://www.enterprisedb.com/downloads/postgres-postgresql-downloads . Therefore, instead use PostgreSQL for CLANGARM32 https://packages.msys2.org/packages/mingw-w64-clang-aarch64-postgresql .
4. For the files in the `plr.zip` file, and the new corresponding target locations, run from the CLANGARM64 command line the program `pg_config`. To find the new location for the `plr.dll` file, run `echo "$(cygpath -wl "$(pg_config --pkglibdir)")"`. To find the new location of `plr.control` file and `.sql files`, run `echo "$(cygpath -wl "$(pg_config --sharedir)"/extension)"`.

Creating the PLR Extension

As of PostgreSQL 9.1 you can use the new `CREATE EXTENSION` command:

```
CREATE EXTENSION plr;
```

This is not only simple, it has the added advantage of tracking all PL/R installed objects as dependent on the extension, and therefore they can be removed just as easily if desired:

```
DROP EXTENSION plr;
```

Tip If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

Tip In addition to the documentation, the plr.out.* files in the plr/expected folder are a good source of usage examples.

Functions and Arguments

To create a function in the PL/R language, use standard R syntax, but without the enclosing braces or function assignment. Instead of `myfunc <- function(arguments) { function body }`, the body of your PL/R function is just `sqlfunction body`

```
CREATE OR REPLACE FUNCTION funcname(argument-types) RETURNS return-type AS '
function body
' LANGUAGE plr;
```

The body of the function is simply a piece of R script. When the function is called, the argument values are passed as variables `arg1...argN` to the R script. The result is returned from the R code in the usual way. For example, a function returning the greater of two integer values could be defined as:

```
CREATE OR REPLACE FUNCTION r_max(integer, integer) RETURNS integer AS '
if (arg1 > arg2)
  return(arg1)
else
```

```
  return(arg2)
' LANGUAGE plr STRICT;
```

Literal characters in the body of an R function that is within the body of a PL/R function can be written in double quotes (“) or in single quotes (‘) (both, just like R), except that in PL/R each single quote is escaped with a preceding single quote(’). Also, in PostgreSQL functions, dollar signs may distinquish the beginning and end of a string boundary. Some examples follow.

```
CREATE OR REPLACE FUNCTION hello() RETURNS text AS '
return(''Hello'')
' LANGUAGE plr;

CREATE OR REPLACE FUNCTION hello2() RETURNS text AS '
return("Hello")
' LANGUAGE plr;

CREATE OR REPLACE FUNCTION hello3() RETURNS text AS
$body$
return('Hello')
$body$ LANGUAGE plr;

CREATE OR REPLACE FUNCTION hello4() RETURNS text AS
$body$
return("Hello")
$body$ LANGUAGE plr;

SELECT hello();

 hello
-------
 Hello
(1 row)

SELECT hello2();

 hello2
--------
 Hello
(1 row)

SELECT hello3();

 hello3
--------
 Hello
(1 row)

SELECT hello4();

 hello4
--------
 Hello
(1 row)
```

Starting with PostgreSQL 8.0, arguments may be explicitly named when creating a function. If an argument is explicitly named at function creation time, that name will be available to your R script in place of the usual `argN variable`. For example:

```
CREATE OR REPLACE FUNCTION sd(vals float8[]) RETURNS float AS '
sd(vals)
' LANGUAGE plr STRICT;
```

Starting with PostgreSQL 8.4, a PL/R function may be declared to be a `WINDOW`. In this case, in addition to the usual `argN(or named)` variables, PL/R automatically creates several other arguments to your function. For each explicit argument, a corresponding variable called `farg1...fargN` is passed to the R script.

These contain an R vector of all the values of the related argument for the moving `WINDOW` frame within the current `PARTITION`. For example:

```
CREATE OR REPLACE FUNCTION r_regr_slope(float8, float8, int) RETURNS float8 AS '
slope <- NA
y <- farg1
x <- farg2
preceding <- arg3
if (fnumrows == preceding + 1L)
  try (slope <- lm(y ~ x)$coefficients[2])
return(slope)
' LANGUAGE plr WINDOW;
```

In the preceding example,`farg1` and `farg2` are R vectors containing the current row's data plus that of related rows. The determination as to which rows qualify as related is determined by the frame specification of the query at run time. The example also illustrates one of two additional autogenerated arguments. `fnumrows` is the number of rows in the current `WINDOW` frame. The other (not shown) auto-argument is called `prownum`. This argument provides the 1-based row offset of the current row in the current `PARTITION`. See Window Functions for more information and a more complete example.

In some of the the definitions above, note the clause `STRICT`, which saves us from having to think about NULL input values: if a NULL is passed, the function will not be called at all, but will just return a NULL result automatically. In a non-strict function, if the actual value of an argument is NULL, the corresponding `argN` variable will be set to a `NULL R` object. For example, suppose that we wanted `r_max` with one null and one non-null argument to return the non-null argument, rather than NULL:

```
CREATE OR REPLACE FUNCTION r_max(integer, integer) RETURNS integer AS '
if (is.null(arg1) && is.null(arg2))
  return(NULL)
if (is.null(arg1))
  return(arg2)
if (is.null(arg2))
  return(arg1)
if (arg1 > arg2)
  return(arg1)
' LANGUAGE plr;
```

As shown above, to return a NULL value from a PL/R function, return `NULL`. This can be done whether the function is strict or not. Composite-type (tuple) arguments are passed to the procedure as R data.frames. The element names of the frame are the attribute names of the composite type. If an attribute in the passed row has the NULL value, it will appear as an `NA` in the frame. Here is an example:

```
CREATE TABLE emp(name text, age int, salary numeric(10,2));
INSERT INTO emp VALUES ('Joe', 41, 250000.00);
INSERT INTO emp VALUES ('Jim', 25, 120000.00);
INSERT INTO emp VALUES ('Jon', 35, 50000.00);

CREATE OR REPLACE FUNCTION overpaid (emp) RETURNS bool AS '
if (200000 < arg1$salary) {
  return(TRUE)
}
if (arg1$age < 30 && 100000 < arg1$salary) {
  return(TRUE)
}
return(FALSE)
' LANGUAGE plr;

SELECT name, overpaid(emp) FROM emp;

name  | overpaid
------+----------
Joe   | t
Jim   | t
Jon   | f
(3 rows)
```

There is also support for returning a composite-type result value:

```
CREATE OR REPLACE FUNCTION get_emps() RETURNS SETOF emp AS '
names <- c(''Joe'',''Jim'',''Jon'')
ages <- c(41,25,35)
salaries <- c(250000,120000,50000)
df <- data.frame(name = names, age = ages, salary = salaries)
```

```
return(df)
' LANGUAGE plr;

SELECT * FROM get_emps();

name  | age | salary
------+-----+-----------
Jim   | 41  | 250000.
Joe   | 25  | 120000.
Jon   | 35  | 50000.
(3 rows)
```

An alternative method may be used to create a function in PL/R, if certain criteria are met. First, the function must be a simple call to an existing R function. Second, the function name used for the PL/R function must match that of the R function exactly. If these two criteria are met, the PL/R function may be defined with no body, and the arguments will be passed directly to the R function of the same name.

For example:

```
CREATE OR REPLACE FUNCTION sd(_float8) RETURNS float AS '
' LANGUAGE plr;

SELECT round(sd('{1.23,1.31,1.42,1.27}'::_float8)::numeric,8);

round
------------
0.08180261
(1 row)
```

Tip Because the function body is passed as an SQL string literal to `CREATE FUNCTION`, you have to escape single quotes and backslashes within your R source, typically by doubling them.

Passing Data Values

The argument values supplied to a PL/R function's script are the input arguments converted to a corresponding R form. See Table 4-1. Scalar PostgreSQL values become single element R vectors. One exception to this are scalar bytea values. These are first converted to R raw type, and then processed by the R unserialize command. One-dimensional PostgreSQL arrays are converted to multi-element R vectors, two-dimensional PostgreSQL arrays are mapped to R matrixes, and three-dimensional PostgreSQL arrays are converted to three-dimensional R arrays. Greater than three-dimensional arrays are not supported. Composite-types are transformed into R data.frames.

Table 4-1. Function Arguments

PostgreSQL type	R type
boolean	logical
int2,int4	integer
int8,float4,float8,money,numeric	numeric
bytea	object
everything else	character

Conversely, the return values are first coerced to R character, and therefore anything that resolves to a string that is acceptable input format for the function's declared return type will produce a result. Again, there is an exception for scalar bytea return values. In this case, the R object being returned is first processed by the R serialize command, and then the binary result is directly mapped into a PostgreSQL bytea datum. Similar to argument conversion, there is also a mapping between the dimensionality of the declared PostgreSQL return type and the type of R object. That mapping is shown in Table 4-2

Table 4-2. Function Result Dimensionality

PgSQL return type	R type	Result	Example
scalar	array,matrix,vector	first column of first row	c(1,2,3) in R returns 1 in PostgreSQL
setof scalar	1D array,greater than 2D array, vector	multi-row, 1 column set	array(1:10) in R returns 10 rows in PostgreSQL

PgSQL return type	R type	Result	Example
scalar	data.frame	textual representation of the first column's vector	data.frame(c(1,2,3)) in R returns 'c(1, 2, 3)'
setof scalar	2D array,matrix,data.frame	#columns > 1, error; #columns == 1,multi-row, 1 column set	(as.data.frame(array(1:10,c in R returns 2 rows of scalar
array	1D array,greater than 3D array,vector	1D array	array(1:8,c(2,2,2,2)) in R returns {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7
array	2D array,matrix,data.frame	2D array	array(1:4,c(2,2)) in R returns {{1,3},{2,4}}
array	3D array	3D array	array(1:8,c(2,2,2)) in R returns {{{1,5},{3,7}},{{2,6},{4,8}
composite	1D array,greater than 2D array,vector	first row, 1 column	array(1:8,c(2,2,2)) in R returns 1 row of scalar
setof composite	1D array,greater than 2D array,vector	multi-row, 1 column set	array(1:8,c(2,2,2)) in R returns 8 rows of scalar
composite	2D array,matrix,data.frame	first row, multi-column	array(1:4,c(2,2)) in R returns 1 row of 2 columns
setof composite	2D array,matrix,data.frame	multi-row, multi-column set	array(1:4,c(2,2)) in R returns 2 rows of 2 columns

Using Global Data

Sometimes it is useful to have some global status data that is held between two calls to a procedure or is shared between different procedures. Equally useful is the ability to create functions that your PL/R functions can share. This is easily done since all PL/R procedures executed in one backend share the same R interpreter. So, any global R variable is accessible to all PL/R procedure calls, and will persist for the duration of the SQL client connection. An example of using a global object appears in the `pg.spi.execp` example, in Database Access and Support Functions.

A globally available, user named, R function (the R function name of PL/R functions is not the same as its PostgreSQL function name; see: R Function Names) can be created dynamically using the provided PostgreSQL function `install_rcmd(text)`. Here is an example:

```
SELECT install_rcmd('pg.test.install <-function(msg) {print(msg)}');

install_rcmd
--------------
OK
(1 row)

CREATE OR REPLACE FUNCTION pg_test_install(text) RETURNS text AS '
pg.test.install(arg1)
' LANGUAGE plr;

SELECT pg_test_install('hello world');

pg_test_install
-----------------
hello world
(1 row)
```

A globally available, user named, R function can also be automatically created and installed in the R interpreter. See: Loading R Modules at Startup PL/R also provides a global variable called `pg.state.firstpass`. This variable is reset to TRUE the first time each `PL/R` function is called, for a particular query. On subsequent calls the value is left unchanged. This allows one or more PL/R functions to perform a possibly expensive initialization on the first call, and reuse the results for the remaining rows in the query.

For example:

```
CREATE TABLE t(f1 int);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
INSERT INTO t VALUES (3);

CREATE OR REPLACE FUNCTION f1() RETURNS int AS '
msg <- paste(''enter f1, pg.state.firstpass is '', pg.state.firstpass)
pg.thrownotice(msg)
if (pg.state.firstpass == TRUE)
  pg.state.firstpass <<- FALSE
msg <- paste(''exit f1, pg.state.firstpass is '', pg.state.firstpass)
pg.thrownotice(msg)
return(0)
' LANGUAGE plr;

CREATE OR REPLACE FUNCTION f2() RETURNS int AS '
msg <- paste(''enter f2, pg.state.firstpass is '', pg.state.firstpass)
pg.thrownotice(msg)
if (pg.state.firstpass == TRUE)
  pg.state.firstpass <<- FALSE
msg <- paste(''exit f2, pg.state.firstpass is '', pg.state.firstpass)
pg.thrownotice(msg)
return(0)
' LANGUAGE plr;

SELECT f1(), f2(), f1 FROM t;
NOTICE: enter f1, pg.state.firstpass is TRUE
NOTICE: exit f1, pg.state.firstpass is FALSE
NOTICE: enter f2, pg.state.firstpass is TRUE
NOTICE: exit f2, pg.state.firstpass is FALSE
NOTICE: enter f1, pg.state.firstpass is FALSE
NOTICE: exit f1, pg.state.firstpass is FALSE
NOTICE: enter f2, pg.state.firstpass is FALSE
NOTICE: exit f2, pg.state.firstpass is FALSE
NOTICE: enter f1, pg.state.firstpass is FALSE
NOTICE: exit f1, pg.state.firstpass is FALSE
NOTICE: enter f2, pg.state.firstpass is FALSE
NOTICE: exit f2, pg.state.firstpass is FALSE

f1  | f2 | f1
----+----+----
0   | 0  | 1
0   | 0  | 2
0   | 0  | 3
(3 rows)

CREATE OR REPLACE FUNCTION row_number2() RETURNS int AS '
if (pg.state.firstpass)
{
  assign(''pg.state.firstpass'', FALSE, env=.GlobalEnv)
  lclcntr<- 1
}
else
  lclcntr<- plrcounter + 1
assign(''plrcounter'', lclcntr, env=.GlobalEnv)
return(lclcntr)
' LANGUAGE plr;

SELECT row_number2(), f1 FROM t;

row_number2 | f1
------------+-----
1           | 1
2           | 2
3           | 3
(3 rows)
```

Database Access and Support

Functions

The following commands are available to access the database from the body of a PL/R procedure, or in support thereof:

Normal Support

`pg.spi.exec(character query)`

Execute an SQL query given as a string. An error in the query causes an error to be raised. Otherwise, the command's return value is the number of rows processed for `INSERT` ,`UPDATE`, or `DELETE` statements, or zero if the query is a utility statement. If the query is a `SELECT` statement, the values of the selected columns are placed in an R data.frame with the target column names used as the frame column names. However, non-numeric columns are **not** converted to factors. If you want all non-numeric columns converted to factors, a convenience function `pg.spi.factor` (described below) is provided.

If a field of a SELECT result is NULL, the target variable for it is set to `NA`. For example:

```
CREATE OR REPLACE FUNCTION test_spi_tup(text) RETURNS SETOF record AS '
pg.spi.exec(arg1)
' LANGUAGE plr;

SELECT * FROM test_spi_tup('SELECT oid, NULL::text as nullcol,
typname FROM pg_type WHERE typname = ''oid'' OR typname = ''text''')
AS t(typeid oid, nullcol text, typename name);

typeid  | nullcol | typename
--------+---------+----------
25      |         | text
26      |         | oid
(2 rows)
```

The NULL values were passed to R as `NA`, and on return to PostgreSQL they were converted back to NULL.

`pg.spi.prepare(character query,integer vector type_vector)`

Prepares and saves a query plan for later execution. The saved plan will be retained for the life of the current backend. The query may use arguments, which are placeholders for values to be supplied whenever the plan is actually executed. In the query string, refer to arguments by the symbols `$1...$n`. If the query uses arguments, the values of the argument types must be given as a vector. Pass `NA` for `type_vector` if the query has no arguments. The argument types must be identified by the type Oids, shown in pg_type. Global variables are provided for this use. They are named according to the convention TYPENAMEOID, where the actual name of the type, in all capitals, is substituted for TYPENAME. A support function, `load_r_typenames()` must be used to make the predefined global variables available for use:

```
SELECT load_r_typenames();


load_r_typenames
------------------
OK
(1 row)
```

Another support function,`r_typenames()` may be used to list the predefined Global variables:

```
SELECT * FROM r_typenames();

typename    | typeoid
------------+---------
ABSTIMEOID  | 702
ACLITEMOID  | 1033
ANYARRAYOID | 2277
ANYOID      | 2276
BITOID      | 1560
BOOLOID     | 16
[...]
TRIGGEROID  | 2279
UNKNOWNOID  | 705
VARBITOID   | 1562
VARCHAROID  | 1043
VOIDOID     | 2278
```

```
XIDOID      | 28
(159 rows)
```

The return value from `pg.spi.prepare` is a query ID to be used in subsequent calls to pg.spi.execp. See `spi_execp` for an example.

`pg.spi.execp(external pointer saved_plan, variable listvalue_list)`

Execute a query previously prepared with `pg.spi.prepare`. `saved_plan` is the external pointer returned by `pg.spi.prepare`. If the query references arguments, a `value_list` must be supplied: this is an R list of actual values for the plan arguments. It must be the same length as the argument type_vector previously given to pg.spi.prepare. Pass `NA` for `value_list` if the query has no arguments. The following illustrates the use of `pg.spi.prepare` and `pg.spi.execp` with and without query arguments:

```
CREATE OR REPLACE FUNCTION test_spi_prep(text) RETURNS text AS '
sp <<- pg.spi.prepare(arg1, c(NAMEOID, NAMEOID))
print(''OK'')
' LANGUAGE plr;

SELECT test_spi_prep('SELECT oid, typname FROM pg_type
WHERE typname = $1 OR typname = $2');

test_spi_prep
---------------
OK
(1 row)

CREATE OR REPLACE FUNCTION test_spi_execp(text, text, text) RETURNS SETOF record AS '
pg.spi.execp(pg.reval(arg1), list(arg2,arg3))
' LANGUAGE plr;

SELECT * FROM test_spi_execp('sp','oid','text') AS t(typeid oid, typename name);
typeid  | typename
--------+----------
25      | text
26      | oid
(2 rows)

CREATE OR REPLACE FUNCTION test_spi_prep2(text) RETURNS text AS '
sp <<- pg.spi.prepare(arg1, NA)
print(''OK'')
' LANGUAGE plr;

SELECT test_spi_prep('SELECT oid, typname FROM pg_type
WHERE typname = ''bytea'' OR typname = ''text''');
test_spi_prep
---------------
OK
(1 row)

CREATE OR REPLACE FUNCTION test_spi_execp(text) RETURNS SETOF record AS '
pg.spi.execp(pg.reval(arg1), NA)
' LANGUAGE plr;

SELECT * FROM test_spi_execp('sp') AS t(typeid oid, typename name);

typeid  | typename
--------+----------
17      | bytea
25      | text
(2 rows)

CREATE OR REPLACE FUNCTION test_spi_prep(text) RETURNS text AS '
sp <<- pg.spi.prepare(arg1)
print(''OK'')
' LANGUAGE plr;

SELECT test_spi_prep('SELECT oid, typname FROM pg_type
WHERE typname = ''bytea'' OR typname = ''text''');
test_spi_prep
---------------
OK
```

```
(1 row)

CREATE OR REPLACE FUNCTION test_spi_execp(text) RETURNS SETOF record AS '
pg.spi.execp(pg.reval(arg1))
' LANGUAGE plr;

SELECT * FROM test_spi_execp('sp') AS t(typeid oid, typename name);
typeid  | typename
--------+----------
17      | bytea
25      | text
(2 rows)
```

NULL arguments should be passed as individual `NA` values in value_list. Except for the way in which the query and its arguments are specified,`pg.spi.execp` works just like `pg.spi.exec`.

`pg.spi.cursor_open(character cursor_name, external pointer saved_plan, variable list value_list)`

Opens a cursor identified by cursor_name. The cursor can then be used to scroll through the results of a query plan previously prepared by pg.spi.prepare. Any arguments to the plan should be specified in arg values similar to `pg.spi.execp`. Only read-only cursors are supported at the moment.

```
plan <- pg.spi.prepare('SELECT * FROM pg_class');
cursor_obj <- pg.spi.cursor_open('my_cursor',plan);
```

Returns a cursor object that be be passed to `pg.spi.cursor_fetch`

`pg.spi.cursor_fetch(external pointer cursor, boolean forward, integer rows)`

Fetches rows from the cursor object previously returned by `pg.spi.cursor_open`. If forward is TRUE, then the cursor is moved forward to fetch at most the number of rows required by the rows parameter. If forward is FALSE, then the cursor is moved backwards at most the number of rows specified. rows indicates the maximum number of rows that should be returned.

```
plan <- pg.spi.prepare('SELECT * FROM pg_class');
cursor_obj <- pg.spi.cursor_open('my_cursor',plan);
data <- pg.spi.cursor_fetch(cursor_obj,TRUE,as.integer(10));
```

Returns a data frame containing the results.

`pg.spi.cursor_close(external pointer cursor)`

Closes a cursor previously opened by `pg.spi.cursor_open`

```
plan <- pg.spi.prepare('SELECT * FROM pg_class');
cursor_obj <- pg.spi.cursor_open('my_cursor',plan);
pg.spi.cursor_close(cursor_obj);
```

`pg.spi.lastoid()`

Returns the OID of the row inserted by the last query executed via `pg.spi.exec` or `pg.spi.execp`, if that query was a single-row INSERT. (If not, you get zero.)

`pg.quoteliteral(character SQL_string)`

Duplicates all occurrences of single quote and backslash characters in the given string. This may be used to safely quote strings that are to be inserted into SQL queries given to `pg.spi.exec` or `pg.spi.prepare`.

`pg.quoteident(character SQL_string)`

Return the given string suitably quoted to be used as an identifier in an SQL query string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case folded). Embedded quotes are properly doubled. This may be used to safely quote strings that are to be inserted into SQL queries given to `pg.spi.exec` or `pg.spi.prepare`.

`pg.thrownotice(character message)`

`pg.throwerror(character message)`

Emit a PostgreSQL `NOTICE` or `ERROR` message.`ERROR` also raises an error condition: further execution of the function is abandoned, and the current transaction is aborted.

`pg.spi.factor(data.frame data)`

Accepts an R `data.frame` as input, and converts all non-numeric columns to `factors`. This may be useful for `data.frames` produced by `pg.spi.exec` or `pg.spi.prepare`, because the PL/R conversion mechanism does **not** do that for you.

RPostgreSQL Compatibility Support

The following functions are intended to provide some level of compatibility between PL/R and RPostgreSQL (PostgreSQL DBI package). This allows, for example, a function to be first prototyped using an R client, and then easily moved to PL/R for production use.

```
dbDriver(character dvr_name)

dbConnect (DBIDriver drv, character user, character password, character host, character dbname,            charac
port, character tty, character options)

dbSendQuery(DBIConnection conn, character sql)

fetch(DBIResult rs,integer num_rows)

dbClearResult(DBIResult rs)

dbGetQuery(DBIConnection conn, character sql)

dbReadTable(DBIConnection conn, character name)

dbDisconnect(DBIConnection conn)

dbUnloadDriver(DBIDriver drv)
```

These functions nominally work like their RPostgreSQL counterparts except that all queries are performed in the current database. Therefore all driver and connection related parameters are ignored, and dbDriver, dbConnect, dbDisconnect, and dbUnloadDriver are no-ops.

PostgreSQL Support Functions

The following commands are available to use in PostgreSQL queries to aid in the use of PL/R functions:

```
plr_version()
```

that displays the PL/R version x.y (but not the patch version x.y.z)

```
SELECT plr_version();

 plr_version
-------------
 8.4
(1 row)
```

```
SELECT * FROM pg_available_extensions WHERE name = 'plr'
```

that displays the PL/R version x.y.z

```
SELECT * FROM pg_available_extensions WHERE name = 'plr';

 name | default_version | installed_version |                            comment
------+-----------------+-------------------+-------------------------------------------------------------------
 plr  | 8.4.7           | 8.4.7             | load R interpreter and execute R script from within a database
(1 row)
```

```
r_version()
```

that displays R version . . .

```
SELECT r_version();

                   r_version
-------------------------------------------------
 (platform,x86_64-w64-mingw32)
 (arch,x86_64)
 (os,mingw32)
 (crt,"ucrt")
 (system,"x86_64, mingw32")
 (status,"")
 (major,4)
 (minor,4.1)
 (year,2024)
 (month,06)
 (day,14)
 ("svn rev",86737)
```

```
 (language,R)
 (version.string,"R version 4.4.1 (2024-06-14 ucrt)")
 (nickname,"Race for Your Life")
(15 rows)
```

`install_rcmd(text R_code)`

Install R code, given as a string, into the interpreter. See Using Global Data for an example.

`reload_plr_modules()`

Force re-loading of R code from the plr_modulestable. It is useful after modifying the contents of plr_modules, so that the change will have an immediate effect.

`plr_singleton_array(float8 first_element)`

Creates a new PostgreSQL array, using element `first_element`. This function is predefined to accept one float8 value and return a float8 array. The C function that implements this PostgreSQL function is capable of accepting and returning other data types, although the return type must be an array of the input parameter type. It can also accept multiple input parameters. For example, to define a `plr_array` function to create a text array from two input text values:

```
CREATE OR REPLACE FUNCTION plr_array (text, text) RETURNS text[] AS
'$libdir/plr', 'plr_array'
LANGUAGE C STRICT;
```

```
SELECT plr_array('hello', 'world');

plr_array
---------------
{hello,world}
(1 row)
```

`plr_array_push(float8[] array, float8 next_element)`

Pushes a new element onto the end of an existing PostgreSQL array. This function is predefined to accept one float8 array and a float8 value, and return a float8 array. The C function that implements this PostgreSQL function is capable of accepting and returning other data types. For example, to define a `plr_array_push` function to add a text value to an existing text array:

```
CREATE OR REPLACE FUNCTION plr_array_push(_text, text) RETURNS text[] AS
'$libdir/plr','plr_array_push'
LANGUAGE C STRICT;
```

```
SELECT plr_array_push(plr_array('hello', 'world'), 'how are you');

plr_array_push
-----------------------------
{hello,world,"how are you"}
(1 row)
```

`plr_array_accum(float8[] state_value,float8 next_element)`

Creates a new array using next_element if state_value is NULL. Otherwise, pushes next_element onto the end of state_value. This function is predefined to accept one float8 array and a float8 value, and return a float8 array. The C function that implements this PostgreSQL function is capable of accepting and returning other data types. For example, to define a `plr_array_accum` function to add an int4 value to an existing int4 array:

```
CREATE OR REPLACE FUNCTION plr_array_accum(_int4, int4) RETURNS int4[] AS
'$libdir/plr','plr_array_accum'
LANGUAGE C;
```

```
SELECT plr_array_accum(NULL, 42);

plr_array_accum
-------------
{42}
(1 row)

SELECT plr_array_accum('{23,35}', 42);

plr_array_accum
-----------------
{23,35,42}
```

```
(1 row)
```

This function may be useful for creating custom aggregates. See Aggregate Functions for an example.

`load_r_typenames()`

Installs datatype Oid variables into the R interpreter as globals. See also `r_typenames` below.

`r_typenames()`

Displays the datatype Oid variables installed into the R interpreter as globals. See Database Access and Support Functions for an example.

`plr_environ()`

Displays the environment under which the Postmaster is currently running. This may be useful to debug issues related to R specific environment variables. This function is installed with EXECUTE permission revoked from PUBLIC.

`plr_set_display(text display)`

Sets the DISPLAY environment variable under which the Postmaster is currently running. This may be useful if using R to plot to a virtual frame buffer. This function is installed with EXECUTE permission revoked from PUBLIC.

`plr_get_raw(bytea serialized_object)`

By default, when R objects are returned as type `bytea`, the R object is serialized using an internal R function prior to sending to PostgreSQL. This function deserializes the R object using another internal R function, and returns the pure raw bytes to PostgreSQL. This is useful, for example, if the R object being returned is a JPEG or PNG graphic for use outside of R.

Aggregate Functions

Aggregates in PostgreSQL are extensible via SQL commands. In general, to create a new aggregate, a state transition function and possibly a final function are specified. The final function is used in case the desired output of the aggregate is different from the data that needs to be kept in the running state value. There is more than one way to create a new aggregate using PL/R. A simple aggregate can be defined using the predefined PostgreSQL C function,`plr_array_accum` (see PostgreSQL Support Functions) as a state transition function, and a PL/R function as a finalizer. For example:

```
CREATE OR REPLACE FUNCTION r_median(_float8) RETURNS float AS '
median(arg1)
' LANGUAGE plr;

CREATE AGGREGATE median (
  sfunc = plr_array_accum,
  basetype = float8,
  stype = _float8,
  finalfunc = r_median
);

CREATE TABLE foo(f0 int, f1 text, f2 float8);
INSERT INTO foo VALUES(1,'cat1',1.21);
INSERT INTO foo VALUES(2,'cat1',1.24);
INSERT INTO foo VALUES(3,'cat1',1.18);
INSERT INTO foo VALUES(4,'cat1',1.26);
INSERT INTO foo VALUES(5,'cat1',1.15);
INSERT INTO foo VALUES(6,'cat2',1.15);
INSERT INTO foo VALUES(7,'cat2',1.26);
INSERT INTO foo VALUES(8,'cat2',1.32);
INSERT INTO foo VALUES(9,'cat2',1.30);

SELECT f1, median(f2) FROM foo GROUP BY f1 ORDER BY f1;

f1    | median
------+--------
cat1  | 1.21
cat2  | 1.28
(2 rows)
```

A more complex aggregate might be created by using a PL/R functions for both state transition and finalizer.

Window Functions

Starting with version 8.4, PostgreSQL supports `WINDOW` functions which provide the ability to perform calculations across sets of rows that are related to the current query row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row; the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result. See the PostgreSQL documentation for more general information related to the use of this capability.

PL/R functions may be defined as `WINDOW`. For example:

```
CREATE OR REPLACE FUNCTION r_regr_slope(float8, float8, int) RETURNS float8 AS '
slope <- NA
y <- farg1
x <- farg2
preceding <- arg3
if (fnumrows == preceding + 1L)
  try (slope <- lm(y ~ x)$coefficients[2])
return(slope)
' LANGUAGE plr WINDOW;
```

A number of variables are automatically provided by PL/R to the R interpreter:

`fargN`

`farg1` and `farg2` are R vectors containing the current row's data plus that of the related rows.

`fnumrows`

The number of rows in the current `WINDOW` frame.

`prownum` (not shown)

Provides the 1-based, row offset of the current row, in the current `PARTITION`.

A more complete example follows:

```
-- CREATE test TABLE
CREATE TABLE test_data (
  fyear integer,
  firm float8,
  eps float8
);

-- insert randomly perturbated data for test
INSERT INTO test_data
SELECT (b.f + 1) % 10 + 2000       AS fyear,
       floor((b.f+1)/10) + 50      AS firm,
       f::float8/100 + random()/10 AS eps
FROM generate_series(-500,499,1) b(f);

CREATE OR REPLACE FUNCTION r_regr_slope(float8, float8, int) RETURNS float8 AS '
slope <- NA
y <- farg1
x <- farg2
preceding <- arg3
if (fnumrows == preceding + 1L)
  try (slope <- lm(y ~ x)$coefficients[2])
return(slope)
' LANGUAGE plr WINDOW;

SELECT *, r_regr_slope(eps, lag_eps, 8) OVER w AS slope_R
FROM (SELECT firm, fyear, eps, lag(eps) OVER (ORDER BY firm, fyear) AS lag_eps
      FROM test_data) AS a
WHERE eps IS NOT NULL
WINDOW w AS (ORDER BY firm, fyear ROWS 8 PRECEDING);
```

In this example, the variables `farg1` and `farg2` contain the current row value for eps and lag_eps, as well as the preceding 8 rows which are also in the same `WINDOW` frame within the same `PARTITION`. In this case since no `PARTITION` is explicitly defined, the `PARTITION` is the entire set of rows returned from the inner sub-select.

In these next examples, use of the variables `arg1`,`farg1`,`fnumrows`, and `prownum` are illustrated in detail. The window frame is saved into a dedicated R Environment.

Variables `farg#` and only `farg1` in these simple examples. The `foo` table that we created (above) is reused here.

```
CREATE OR REPLACE FUNCTION arg1(int) RETURNS int AS '
return(arg1)
' LANGUAGE plr WINDOW;

CREATE OR REPLACE FUNCTION farg1(int) RETURNS text AS '
return(capture.output(farg1))
' LANGUAGE plr WINDOW;

CREATE OR REPLACE FUNCTION fnumrows(int) RETURNS int AS '
return(fnumrows)
' LANGUAGE plr WINDOW;

CREATE OR REPLACE FUNCTION prownum(int) RETURNS int AS '
return(prownum)
' LANGUAGE plr WINDOW;

SELECT s.f1, s.f0,
           arg1(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0),
          farg1(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0),
       fnumrows(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0),
        prownum(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0)
FROM (SELECT f0 + 10 f0, f1 FROM foo) AS s;

  f1  | f0 | arg1 |       farg1        | fnumrows | prownum
------+----+------+--------------------+----------+---------
 cat1 | 11 |   11 | [1] 11             |        1 |       1
 cat1 | 12 |   12 | [1] 11 12          |        2 |       2
 cat1 | 13 |   13 | [1] 11 12 13       |        3 |       3
 cat1 | 14 |   14 | [1] 11 12 13 14    |        4 |       4
 cat1 | 15 |   15 | [1] 11 12 13 14 15 |        5 |       5
 cat2 | 16 |   16 | [1] 16             |        1 |       1
 cat2 | 17 |   17 | [1] 16 17          |        2 |       2
 cat2 | 18 |   18 | [1] 16 17 18       |        3 |       3
 cat2 | 19 |   19 | [1] 16 17 18 19    |        4 |       4
(9 rows)

SELECT s.f1, s.f0,
           arg1(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0 ROWS 1 PRECEDING),
          farg1(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0 ROWS 1 PRECEDING),
       fnumrows(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0 ROWS 1 PRECEDING),
        prownum(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0 ROWS 1 PRECEDING)
FROM (SELECT f0 + 10 f0, f1 FROM foo) AS s;

  f1  | f0 | arg1 |   farg1    | fnumrows | prownum
------+----+------+------------+----------+---------
 cat1 | 11 |   11 | [1] 11     |        1 |       1
 cat1 | 12 |   12 | [1] 11 12  |        2 |       2
 cat1 | 13 |   13 | [1] 12 13  |        2 |       3
 cat1 | 14 |   14 | [1] 13 14  |        2 |       4
 cat1 | 15 |   15 | [1] 14 15  |        2 |       5
 cat2 | 16 |   16 | [1] 16     |        1 |       1
 cat2 | 17 |   17 | [1] 16 17  |        2 |       2
 cat2 | 18 |   18 | [1] 17 18  |        2 |       3
 cat2 | 19 |   19 | [1] 18 19  |        2 |       4
(9 rows)
```

Programmer-created temporary (or utility) variables and their values may be needed by the user need to be re-available within the next call of the function, that is, at the next position of the row pointer. These variables can be stashed in the R environment `parent.frame(0)`. Do not stash these variables in the R global environment `.RGlobalEnv`.

```
CREATE OR REPLACE FUNCTION framefirst_plus_current(int) RETURNS int AS '
if(prownum == 1)
  assign(''frame_first_value'', arg1, envir = parent.frame())
```

```
return(frame_first_value + farg1[fnumrows])
' LANGUAGE plr WINDOW;

SELECT s.f1, s.f0,
        framefirst_plus_current(s.f0) OVER(PARTITION BY s.f1 ORDER BY f0)
FROM (SELECT f0 + 10 f0, f1 FROM foo) AS s;

  f1  | f0 | framefirst_plus_current
------+----+-------------------------
 cat1 | 11 |                      22
 cat1 | 12 |                      23
 cat1 | 13 |                      24
 cat1 | 14 |                      25
 cat1 | 15 |                      26
 cat2 | 16 |                      32
 cat2 | 17 |                      33
 cat2 | 18 |                      34
 cat2 | 19 |                      35
(9 rows)
```

Another interesting example follows. The idea of "Winsorizing" is to return either the original value or, if that value is outside certain bounds, a trimmed value. So for example `winsorize(eps, 0.1)` would return the value at the 10th percentile for values of eps less that that, the value of the 90th percentile for eps greater than that value, and the unmodified value of eps otherwise.

Everytime, the row pointer is moved, to prevent the re-calcuation of the 'frame result' the to-be-once calculated frame_result, that is only needs to be calculated at `prownum == 1`, is stored in the parent.frame.

```
CREATE OR REPLACE FUNCTION winsorize(float8, float8) RETURNS float8 AS '
if(prownum == 1L)
  assign(''frame_result'', psych::winsor(as.vector(farg1), arg2), envir = parent.frame())
return(frame_result[prownum])
' LANGUAGE plr VOLATILE WINDOW;
```

Here is the example call through SQL. Note, the R CRAN package `psych` (and the dependencies), must have already been installed into R.

```
SELECT fyear, eps,
winsorize(eps, 0.1) OVER (PARTITION BY fyear) AS w_eps
FROM test_data ORDER BY fyear, eps;
```

For optimization reasons, constant expressions are not expanded.

The corresponding `farg2` in the `Winsorize` example above is passes with NULL value. Compatibility reasons exist, so that other arguments are not shifted, in functions users created with previous versions of PL/R.

Loading R Modules at Startup

PL/R has support for auto-loading R code during interpreter initialization. It uses a special table, `plr_modules`, which is presumed to contain modules of R code. If this table exists, the modules defined are fetched from the table and loaded into the R interpreter immediately after creation. The definition of the table `plr_modules` is as follows:

```
CREATE TABLE plr_modules (
  modseq int4,
  modsrc text
);
```

The column `modseq` is used to control the order of installation. The column `modsrc` contains the full text of the R code to be executed, including assignment if that is desired. Consider, for example, the following statement:

```
INSERT INTO plr_modules
VALUES (0, 'pg.test.module.load <-function(msg) {print(msg)}');
```

This statement will cause an R function namedpg.test.module.load to be created in the R interpreter on initialization. A PL/R function may now simply reference the function directly as follows:

```
CREATE OR REPLACE FUNCTION pg_test_module_load(text) RETURNS text AS '
pg.test.module.load(arg1)
' LANGUAGE plr;
```

```
SELECT pg_test_module_load('hello world');

pg_test_module_load
--------------------
hello world
(1 row)
```

The table `plr_modules` must be readable by all, but it is wise to make it owned and writable only by the database administrator.

R Function Names

In PostgreSQL, a function name can be used for different functions (overloaded) as long as the number of arguments or their types differ. R, however, requires all function names to be distinct. PL/R deals with this by constructing the internal R function names as a concatenation of the string "PLR" with the object ID of the procedure's `pg_proc`. Thus, PostgreSQL functions with the same name and different argument types will be different R functions too. This is not normally a concern for a PL/R programmer, but it might be visible when debugging. If a specific, known, function name is needed so that an R function can be referenced by one or more PL/R functions, the `install_rcmd(text)` command can be used. See Using Global Data.

Trigger Procedures

Trigger procedures can be written in PL/R. PostgreSQL requires that a procedure that is to be called as a trigger must be declared as a function with no arguments and a return type of `trigger`. The information from the trigger manager is passed to the procedure body in the following variables:

`pg.tg.name`

The name of the trigger from the `CREATE TRIGGER` statement.

`pg.tg.relid`

The object ID of the table that caused the trigger procedure to be invoked.

`pg.tg.relname`

The name of the table that caused the trigger procedure to be invoked.

`pg.tg.when`

The string `BEFORE` or `AFTER` depending on the type of trigger call.

`pg.tg.level`

The string `ROW` or `STATEMENT` depending on the type of trigger call.

`pg.tg.op`

The string `INSERT`,`UPDATE`, or `DELETE` depending on the type of trigger call.

`pg.tg.new`

When the trigger is defined `FOR EACH ROW`, a data.frame containing the values of the new table row for `INSERT` or `UPDATE` actions. For triggers defined `FOR EACH STATEMENT` and for `DELETE` actions, set to `NULL`. The attribute names are the table's column names. Columns that are null will be represented as `NA`.

`pg.tg.old`

When the trigger is defined `FOR EACH ROW`, a data.frame containing the values of the old table row for `DELETE` or `UPDATE` actions. For triggers defined `FOR EACH STATEMENT` and for `INSERT` actions, set to `NULL`. The attribute names are the table's column names. Columns that are null will be represented as `NA`.

`pg.tg.args`

A vector of the arguments to the procedure as given in the `CREATE TRIGGER` statement. The return value from a trigger procedure can be `NULL` or a one row data.frame matching the number and type of columns in the trigger table. `NULL` tells the trigger manager to silently suppress the operation for this row. If a one row data.frame is returned, it tells PL/R to return a possibly modified row to the trigger manager that will be inserted instead of the one given in `pg.tg.new`. This works for `INSERT` and `UPDATE` only. Needless to say that all this is only meaningful when the trigger is r`BEFORE` and `FOR EACH ROW`; otherwise the return value is ignored.

Here's a little example trigger procedure that forces an integer value in a table to keep track of the number of updates that are performed on the row. For new rows inserted, the value is initialized to 0 and then incremented on every update operation.

```
CREATE TABLE mytab(num integer, description text, modcnt integer);
```

Notice below, that the trigger procedure itself does not know the column name; that's supplied from the trigger arguments. This lets the trigger procedure be reused with different tables.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS '
if (pg.tg.op == ''INSERT'')
{
  retval <- pg.tg.new
  retval[pg.tg.args[1]] <- 0
}
if (pg.tg.op == ''UPDATE'')
{
  retval <- pg.tg.new
  retval[pg.tg.args[1]] <- pg.tg.old[pg.tg.args[1]] + 1
}
if (pg.tg.op == ''DELETE'')
  retval <- pg.tg.old
return(retval)
' LANGUAGE plr;

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');

INSERT INTO mytab(num, description) VALUES(11, 'eleven');


SELECT * FROM mytab;

 num | description | modcnt
-----+-------------+--------
  11 | eleven      |      0
(1 row)


INSERT INTO mytab(num, description) VALUES(12, 'twelve');


SELECT * FROM mytab;

 num | description | modcnt
-----+-------------+--------
  11 | eleven      |      0
  12 | twelve      |      0
(2 rows)


UPDATE mytab SET description = 'twelve again' WHERE num = 12;


SELECT * FROM mytab;

 num | description  | modcnt
-----+--------------+--------
  11 | eleven       |      0
  12 | twelve again |      1
(2 rows)


DELETE FROM mytab WHERE num = 12;


SELECT * FROM mytab;

 num | description | modcnt
-----+-------------+--------
  11 | eleven      |      0
(1 row)
```

Inline Handler

In PL/R version 8.4, is the `DO` inline handler. The DO inline handler allows the execution of an anonymous PL/R code block.

```
SELECT plr_version();
 plr_version
-------------
 8.4

DO LANGUAGE plr '
pg.throwlog(''Hello, world!'')
';
```

Output is seen in the PostgreSQL log:

```
2022-07-20 20:05:04.017 UTC [43852] LOG:  Hello, world!

DO LANGUAGE plr '
pg.thrownotice(''Hello, world!'')
';
```

Output is seen in the user console:

```
NOTICE:  Hello, world!
DO
```

Stored Procedures

In PostgreSQL version eleven(11) or later, is the feature of Stored Procedures. These work in any operating system and in any platform.

```
SELECT version();
                           version
-------------------------------------------------------------
 PostgreSQL 11.0, compiled by Visual C++ build 1914, 64-bit
(1 row)

SELECT current_setting('server_version_num')::int;

 current_setting
-----------------
          110000
(1 row)
```

Unlike functions that return a value, procedures do not return a value.

```
CREATE TABLE tbl(val integer);

CREATE OR REPLACE PROCEDURE insert_data(a int, b int) AS '
pg.spi.exec(''INSERT INTO tbl VALUES (1);'')
pg.spi.exec(''INSERT INTO tbl VALUES (2);'')
' LANGUAGE plr;

CALL insert_data(1, 2);

SELECT * FROM tbl;

 val
-----
   1
   2
(2 rows)
```

Transactions in Stored Procedures

This feature has the same PostgreSQL version requirement as seen in Stored Procedures. Also PL/R version 8.4.2 (or later) is required.

```
CREATE TABLE test1 (a int, b text);

CREATE OR REPLACE PROCEDURE transaction_test1() AS '
  for(i in 0:9)
```

```
  {
    pg.spi.exec(paste(''INSERT INTO test1 (a) VALUES ('', i, '');''))
    if (i %% 2 == 0)
    {
      pg.spi.commit()
    } else {
      pg.spi.rollback()
    }
  }
' LANGUAGE plr;

CALL transaction_test1();

SELECT * FROM test1;

 a | b
---+---
 0 |
 2 |
 4 |
 6 |
 8 |
(5 rows)
```

Custom Type (Tuple) Arguments in Window Functions

Arguments now can be a custom tuple or a record. PL/R version 8.4.2 (or later) is required.

```
CREATE OR REPLACE FUNCTION fast_win_frame(r int, t record) RETURNS bool AS '
identical(parent.frame(), .GlobalEnv) &&
  pg.throwerror(''Parent env is global'')
exists(''plr_window_frame'', parent.frame(), inherits=FALSE) ||
  pg.throwerror(''No window frame data found'')
r == farg2[[prownum, 2]][3]
' LANGUAGE plr WINDOW;

SELECT s.r, s.p, fast_win_frame(NULLIF(r,4), (s.r, s.q)) OVER w
FROM (SELECT r, r % 2 AS p, array_fill(CASE WHEN r=7 THEN 77 ELSE r END, ARRAY[3]) AS q
      FROM generate_series(1,10) r) s
WINDOW w AS (PARTITION BY p ORDER BY r ROWS BETWEEN UNBOUNDED PRECEDING AND
                                                    UNBOUNDED FOLLOWING)
ORDER BY s.r;

 r  | p | fast_win_frame
----+---+----------------
  1 | 1 | t
  2 | 0 | t
  3 | 1 | t
  4 | 0 |
  5 | 1 | t
  6 | 0 | t
  7 | 1 | f
  8 | 0 | t
  9 | 1 | t
 10 | 0 | t
(10 rows)
```

License

License: GPL version 2 or newer. http://www.gnu.org/copyleft/gpl.html This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this

program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA