

# Embedding R in Postgres

Duncan Temple Lang

April 11, 2001

## Abstract

We describe a mechanism by which the R statistical environment is embedded in the Postgres Relational Database Management System. The result is that one can use existing functions and define new functions in the R programming language and invoke them as regular, first-class SQL functions within queries from different clients. The R engine can be used for regular, per-record functions, aggregate functions that operate on collections of records and trigger or event functions that are invoked when operations are performed on a table. The benefits of using R within Postgres include

- immediate access to existing, comprehensive, sophisticated and stable statistical functionality and new functionality added to R as it becomes available;
- access to statistical functionality from within the familiar SQL environment of the clients, without the need for users to learn a statistical environment.
- the ability to send code to the database from a client, rather than transmitting data from database to the client and performing the computations there. This reduces bandwidth and avails of the powerful resources usually associated with database servers.

## Contents

<b>1</b>	<b>Per-Record Transformation Functions</b>	<b>1</b>
<b>2</b>	<b>Aggregates</b>	<b>2</b>
<b>3</b>	<b>Example</b>	<b>4</b>
<b>4</b>	<b>Triggers</b>	<b>8</b>
<b>5</b>	<b>Conversion of Values between Postgres and R</b>	<b>9</b>
<b>6</b>	<b>NOTIFY and LISTEN</b>	<b>9</b>
<b>7</b>	<b>Running the Postgres Server</b>	<b>9</b>
<b>8</b>	<b>Converter Tests</b>	<b>9</b>

## 1 Per-Record Transformation Functions

Many SQL functions operate on each record in a query and compute some transformation of one or more of its values. For example, most RDBMS provide the basic mathematical functions such as pow, sin, log, etc. ; string functions such as concat, substring, upper; network functions such as host, ipaddr, ... We can use R functions to implement new SQL functions in Postgres.

```
<>≡ 2a>  
CREATE FUNCTION gamma(float8) RETURNS float8 AS '' LANGUAGE 'pl_R';
```

```

2a <>+≡ < 2b>
    postgres=# select gamma(10);
           gamma
    -----
           362880
    (1 row)

```

Rather than using existing functions, we can define new R functions and register them as Postgres functions. We define these in the R procedural programming language *pIR*, specifying the function as the value of the **AS** clause in a **CREATE FUNCTION** command.

```

2b <>+≡ <2a 2c>
    CREATE FUNCTION gamma100(float8) RETURNS float8 AS 'function(x) { gamma(x/100)}' LANGUAGE

```

We can then call this Postgres function using an SQL statement.

```

2c <>+≡ <2b 2d>
    select gamma100(joint1_width) from flea;

```

Note that the function is invoked in Postgres and passed to R for execution. Postgres converts each of the function's arguments to an R object and then evaluates the function.

## 2 Aggregates

Aggregate functions in Postgres are used to compute a single value or object (e.g. an array, a user specified type, etc.) from a collection of records. Examples of such aggregate functions are sum, average, SD, max. The Postgres engine processes a query and identifies the resulting records in that query. For each record, it calls the aggregate function identified in that query. That function updates its internal state and returns control back to the Postgres query engine. When all the records have been passed to the function, the Postgres engine, a final or wrapup function is called that extracts the result.

A natural implementation of aggregate functions is to use objects which maintain their own state. For example, to compute the maximum of a collection of numbers, we would create an object that has a field to store the largest value of all the records seen so far. As each new value is passed to the aggregate function, we update the maximum value stored in the object.

In R, we can do this conveniently with a closure. We define a local variable that is visible to the two functions (*update()* and *getMaximum()*) defined within the closure. The *update()* function is called with the value of each record. The second function – *getMaximum()* – provides access to the final value.

```

2d <>+≡ <2c 3a>
    rmax <- function() {
      maxVal <- -Inf
      update <- function(record) {
        maxVal <-<- max(record, maxVal)
      }
      getMaximum <- function()
        maxVal

      return(list(update = update, getMaximum = getMaximum))
    }

```

We can use the R function `rmax()` from Postgres by registering a suitable Postgres function that calls this. We do this using the `CREATE AGGREGATE` command in Postgres. This expects 5 arguments or, more specifically, attributes.

**basetype** The first is *basetype* and is the type of record value on which the aggregator function expects to work. In our case, we are dealing with real numbers, so we specify `float8`.

**stype1** Postgres creates and manages an object that is used to maintain the state of the computation across different records and invocations of the per-record aggregator function. The `stype1` attribute specifies the type that Postgres should create. For R aggregator functions, this is the type `RAggregator` introduced in this R-Postgres procedural language. This is implemented internally (very simply) in C and is a valid Postgres type. It can be used as a parameter and return type of any Postgres function. It has input and output methods. The input method takes an R expression and evaluates it. The resulting R object is stored internally in the `RAggregator` object.

**initcond1** Specifying an `RAggregator` as the basic and generic type for all R aggregation functions wouldn't be much use unless we were able to parameterize the calls with different R types. We can do this by specifying an initialization expression for the `RAggregator` object being used in each aggregation function. This is a string that is an R script or expression that is evaluated to create a suitable R object. In our `rmax()` example, assuming that the closure generator is already defined within R (e.g. via the **Rprofile** script), we can create an instance of the closure definition.

```
3a  (>)+≡                                                                    <2d 3b>
      initcond1 = 'rmax()'
```

Alternatively, we can even define the function within this string and invoke it.

```
3b  (>)+≡                                                                    <3a 3c>
      initcond1 = '(function() {
        maxVal <- -Inf
        update <- function(record) {
          maxVal <<- max(record, maxVal)
        }
        getMaximum <- function()
          maxVal

        return(list(update = update, getMaximum = getMaximum))
      })()'
```

**sfunc1** The first three attributes indicate the type of data on which the function works and what type of R function will perform the aggregation. Next, we specify which *Postgres* function should be called as the state-transition function, or in other words, the per-record call. We have provided functions for the basic Postgres types (e.g. `float8`, `int4`). These are given in table ???. These are functions defined to be implemented in the procedural language *plaggregate*. This is a specialized version of the R procedural language. It is used for aggregate functions only and expects the first argument to be an `RAggregator`. To define a new function of this form that has a different second argument, simply issue the command

```
3c  (>)+≡                                                                    <3b >
      CREATE FUNCTION rsum_add(RAggregator, <your own type>) returns OPAQUE AS '' LANGUAGE
```

You do not have to implement this function. The *plaggregate* handler will automatically convert the argument value to the appropriate type. The command merely declares the function types and allows Postgres to use this information when converting values to the appropriate types and allows the handlers to query the information when attempting to convert the values to R objects.

Note that, *currently*, you can use any name when defining such a function. Since it is an element of the *plaggregate* procedural language, that language's handler will be invoked. At present, this handler ignores the name of the function and simply invokes the first function in the R closure stored within the RAggregator. In future, we might use the name of the Postgres function to identify the entry in the closure list. Additionally, we might use the contents of the **AS** clause (if specified by the user) as either the index or the function definition.

**finalfunc** Having iterated over all the records, invoking the state transition function specified by the *sfunc1*, we need to provide a function for retrieving the answer from the RAggregator. This attribute allows us to specify this function which expects a single argument of type RAggregator. What complicates this attribute is that we have to specify a function that has the correct return type. Again, we have provided Postgres functions (and the corresponding C routines) for the basic types. One can introduce new ones.

It is perhaps simplest to define a new procedural language and handler that calls the appropriate wrapup function on the R object and then converts the answer to the specified type. In other words, we would have a procedural language, say *pl'aggregate'final*, and a corresponding C routine registered as its handler. This routine would call the R function on the R object retrieved from the RAggregator passed to it to get the result. Then it would use the conversion mechanisms used in the regular *plr* procedural language to examine the expected return type of the Postgres function being called and convert the R result object to the appropriate Postgres type. Postgres users would then only have to declare this final function rather than have to implement it.

This is now done and named *pl'aggregate'final*.

### 3 Example

Given the above, we can now illustrate how to define a new aggregate function in Postgres using R. We will again use an example that is already present in Postgres - sum. The command to register the Postgres function is the following:

```
<)+≡ <3c 5a>
CREATE AGGREGATE rsum (
    sfunc1 = r_update_float8,
    basetype = float8,
    stype1 = RAggregator,
    finalfunc = pl_ragg_float8_result,
    initcond1 = '(function() {
total <- 0
add <- function(val) {
    total <-< total + val
}

getTotal <- function() {
    total
}

return(list(add = add, getTotal = getTotal))
})()'
);
```

We can then use this function as a regular Postgres function, without concern for how it is implemented. It can be part of a query such as

```
5a  (>)+≡ < 5b>
      select rsum(joint1_width) from flea;
```

where flea is a table in the database. (It is a dataset distributed with xgobi.)

Let's go back to the command above that defines the function rsum.

```
5b  (>)+≡ <5a 6a>
      CREATE FUNCTION rmax_add(RAggregator, float8) RETURNS RAggregator AS '' LANGUAGE 'pl_R_ag
      CREATE AGGREGATE rmax (
        sfunc1 = rmax_add,
        basetype=float8,
        stype1 = RAggregator,
        finalfunc = pl_ragg_float8_result,
        initcond1 = '(function() {
          maxVal <- -Inf
          update <- function(record) {
            maxVal <<- max(record, maxVal)
          }
          getMaximum <- function()
            maxVal

          return(list(update = update, getMaximum = getMaximum))
        })()'
      );
```

When using a closure, we can specify which element within the list of functions in that closure should be used for update and which should be used for retrieving the value. There are several different ways to do this. The first convention is used above: the first element in the list is the update function and the second element is the result-retrieval function. On occasions, we will be using 3<sup>rd</sup> party code that produces the R object that is to be used as the aggregator. If the update and result functions are not in the first and second positions, we can permute them so that they are. Since we can provide the R statement that creates the aggregator, we can reorder the elements.

```
6a (<)+≡
CREATE FUNCTION rupdate(RAggregator, float8) RETURNS RAggregator AS '' LANGUAGE 'pl_R_agg' <5b 6b>

CREATE AGGREGATE rmin (
  sfunc1 = rupdate,
  basetype=float8,
  stype1 = RAggregator,
  finalfunc = pl_ragg_float8_result,
  initcond1 = '(function() {
    minVal <- Inf
    getMinimum <- function() {
      minVal
    }

    update <- function(record) {
      minVal <<- min(record, minVal)
    }

    return(list(rupdate = update, getMinimum = getMinimum))
  })()[2,1]'
);
```

An alternative is to use the names of the Postgres functions registered given as the values of *sfunc1* and *finalfunc*.

We can declare a function as a member of the *plRaggregate* language that takes the *RAggregator* and the record value. When this function is specified as the value for *sfunc1*, the *plRaggregate* language handler looks for an element in the R list stored in the internal *CRAggregator* and uses that as the update function. If no such element is found in the R object, then the first element is used, as above.

```
6b (<)+≡
CREATE FUNCTION rupdate(RAggregator, float8) RETURNS RAggregator AS '' LANGUAGE 'pl_R_ragg' <6a >
```

Similarly, one can declare a function as a member of the *pl'r wrapup* procedural language collection and give this as the value of the *finalfunc*. The handler for that procedural language will then use the name of this to find an element in the R list. If such an element is found, it is used as the function that returns the result of the aggregation. If no such element is found, the second element of the list is used, as above.

```

<)+≡
CREATE FUNCTION getMinimum(RAggregator) RETURNS float8 AS '' LANGUAGE 'pl_R_wrapup';
                                     <6b >

CREATE AGGREGATE rmin (
  sfunc1 = rupdate,
  basetype=float8,
  stype1 = RAggregator,
  finalfunc = pl_ragg_float8_result,
  initcond1 = '(function() {
    minVal <- Inf
    getMinimum <- function() {
      minVal
    }

    update <- function(record) {
      minVal <<- min(record, minVal)
    }

    return(list(getMinimum = getMinimum, rupdate = update))
  })()'
);

```

One of the benefits of this approach is that we can implement things in a more general way and allow the aggregator to not be implemented as an R list, but instead have it call top-level R functions. None of this really makes a huge difference.

## 4 Triggers

Trigger functions must be defined as taking no arguments and returning OPAQUE. The R function that is called is given a single object of class *PostgresTriggerData*. This contains information about the event that cause the invocation of the trigger function and the arguments specified in the definition of the trigger. This object currently has 7 elements.

**trigger name** this is the name of the trigger being invoked. This is not the name of the function, but the name given for the trigger in the SQL command CREATE TRIGGER name.

**Postgres Function name** the name of the postgres function being called. This might be useful at the R level so we include it here.

**args** This is a character vector containing the values specified as the arguments given in the final element of the registration command for the trigger.

**before** a logical value indicating whether the trigger function is being called before (TRUE) the operation has been performed or after (FALSE).

**row** a logical value indicating whether this trigger invocation was initiated by a row operation (TRUE) or a statement operation (FALSE). This corresponds to being called per-record or per-statement. (Note that in Postgres 7.1, the statement triggers will be removed.)

**op** a string identifying the reason the trigger was called, specifically the type of action that cause the trigger invocation. The possible values are insert, update, delete.

**tuple** this is now an opaque object that contains references to the C-level tuple that triggered the event. One can extract and assign individual values in the tuple, either by name or index (1-based counting) using the usual *R* *[()* and *[j-()* operators (and *setTupleElements()*). Also, one can query the names of the elements in the tuples using *names()*. All the values can be retrieved in a single call as a named *list* via the function *tupleValues()*. Similarly, *tupleTypes()* returns a named character vector describing the Postgres type of each of the elements. *It is vital that the contents of this object are not operated on directly. The values are addresses of C-level objects and modifying them will most likely crash the Postgres server.*

Any changes assigned to elements of the tuple cause the references to change. The new reference(s) is automatically inserted into the R object. This tuple reference must be returned to Postgres as the value of the function if such changes are to be recognized by the Postgres engine.

The trigger can return either the value of the tuple passed to it from Postgres or alternatively NULL. In the latter case (i.e. returning NULL), a NULL tuple will be passed to Postgres and this will be used to finish the operation that led to the trigger invocation. If this is an **INSERT** command, the tuple will not be inserted in this case. In this way, the R function can cause the new tuple to be discarded or dropped. The examples illustrate how this might be use to bound the values of a variable.

If the tuple reference is returned, Postgres takes this value and uses it to complete the event. It is vital that the value of the tuple element in the argument to the function, or any version of it that results from assigning values to any of its elements is returned.

`<>+≡`

```
CREATE FUNCTION r_trigger_test() RETURNS OPAQUE AS 'function(x) {print(x)}' LANGUAGE 'pl'
```

```
CREATE TRIGGER r_trig_test BEFORE INSERT OR DELETE OR UPDATE ON flea FOR EACH ROW EXECUTE PROCEDURE r_trigger_test (1, 'x', 'y', 3.2);
```

```
DROP TRIGGER r_trig_test ON flea;
```

```
INSERT INTO flea VALUES (1,21,1,1,1,1.3);
```

```
CREATE FUNCTION rtupletest() RETURNS OPAQUE AS '' LANGUAGE 'pl_R';
```

## 5 Conversion of Values between Postgres and R

As with all inter-system interfaces, the conversion of values from one system to the other and back again is one of the most important aspects. For environments such as R and Postgres, this process is complicated by the fact that users of each system can define new data types. This makes providing appropriate converters for all data types not only difficult, but (almost) impossible.

## 6 NOTIFY and LISTEN

## 7 Running the Postgres Server

One must set `LD_LIBRARY_PATH` `R_PROFILE`. And also, set `EDITOR` to something. The `R_PROFILE` identifies the name of a file that contains an R script that is to be read and executed when the R engine is started. This is responsible for loading the embedded Postgres library and make its functions available to the R engine. This is needed for trigger functions so that they can access and modify tuple references.

The `LD_LIBRARY_PATH` must be set so that the R shared library `libR.so` is found. This is usually located in `R_HOME/bin`.

## 8 Converter Tests

*(Converter Tests)*≡

```
create function rtext_test(text) returns text AS 'function(x) {paste(x, x)}' LANGUAGE 'p
select rtext_test('My string');
```

```
CREATE FUNCTION r_multiarg_test(int4, text, float8) RETURNS float8 AS 'function(x,y,z) {
GUAGE 'plr';
```

```
select r_multiarg_test(1, 'My string', 3.2);
```

Old stuff.

The per-record Postgres aggregator functions are called with 2 arguments. The second is the value of the record. The first is the object which maintains the state across records. When using R functions as Postgres aggregators, the R object that is used is identified in Postgres as being of type `RAggregator`. This is a regular Postgres type that has input and output methods. The input method is an R expression that is evaluated and expected to return a valid R object. In the case of the `rmax()` function, this should be a call to the closure definition function which returns a list containing the R functions `update()` and `getMaximum()` that share the same instance of `maxVal()`.

this is an object containing the values in the tuple which is being inserted, delete or updated. This is a list of length 2. The first element is a named list of values. The names are the field names of the Postgres table. The second element of this list is a character vector parallel to the values. This gives the names of the Postgres types (e.g float8, text, ...).

```
<  
>  
<Converter Tests  
>
```