

Scenarios for Using R within a Relational Database Management System Server

Duncan Temple Lang
Bell Labs

April 12, 2001

Abstract

We describe an approach for performing general statistical analysis and computations directly within a relational database server rather than the more typical approach of transferring data to the client from the server and having the client perform the analyses. We outline some of the advantages of embedding the R statistical environment (language, interpreter and libraries) within the Postgres server over the usual client-processing approach. We give examples of the three classes of functions currently supported by this embedding approach.

1 Introduction

The reliance on databases has increased dramatically in the past few years. This is likely to increase due to the continuing escalation of the volume of data that we acquire and store. Statisticians have recognized the urgent need to communicate with these Relational Database Management Systems (RDBMS) and improved support has been added to environments such as R[2], and integration into the statistical language in the OmegaHat language. However, support within these languages focus on access for statisticians and data analysts. It is more common that non-data analysts will be accessing the RDBMS servers and would benefit from access to statistical methodology. This is an inversion of the classical way we as statisticians have thought about statistical methodology and RDBMS. We think of embedding access to the RDBMS in the statistical software. We are suggesting embedding statistical software within the RDBMS.

In programming terms, the difference in the approach we discuss here can be succinctly illustrated in the two pseudo commands

```
f( SELECT  x  FROM table )  client-side
SELECT f(x) FROM table    server-side
```

In the first case, the data is returned to the client via the SELECT query, and then the client operates on the values via the function $f()$. In the second approach, the server performs the per-value processing and returns just the result.

The obvious benefits from making statistical software directly accessible from within RDBMS servers include

- users can employ the familiar SQL language to invoke functions implemented in the statistical language without needing to understand that these functions are not built-in SQL functions;
- statisticians can rapidly develop and make available new methodology using their familiar;
- performance improvements accrue since we do not transfer the data to the client and process there, but perform the data reduction on the server and transfer only that reduced data.
- from regular clients, we can add new code to the statistical environment within the server
- there is increased potential for caching results that would ordinarily be computed in different client applications and thus reducing query time.

As with most ideas these days, embedding statistical software within an RDBMS is not an approach that dominates the other. There are numerous trade-offs that have to be understood before deciding on a strategy. The benefit of the approach is that “we” – accessors of data interested in statistical methodology – have more options from which to choose the most appropriate solution. The immediate benefit of embedding R within Postgres [1] is that we can quickly explore these benefits and understand the trade-offs using available software. One potential reason that we may not see performance gains is the vectorized nature of S computations. Most S functions work on multiple observations in a single invocation and are efficient by passing all of the observations to native (C or Fortran) code in one step. When we use S functions for record-wise operations in Postgres, each invocation of the S function will be given just one observation. These repeated calls may incur an overhead and lose the efficiency of the vectorized operations. More investigation is needed.

In this short paper, we will outline three different scenarios in which using statistical software embedded within the server might be useful. In section 2, we show how we can use a fixed model to predict the values for a collection of records within a database. In section 3, we show how we can use the statistical environment for fitting statistical models to large quantities of data using incremental, or record-wise, algorithms. Finally, we show how we can use “trigger functions” to process data as it is entered into the database. This allows us to discard certain observations, and correct or impute fields within the record before they are stored within a table.

These examples illustrate the three different categories of functions that we can define in an SQL extension language. These are record-wise, aggregate, and trigger functions.

While our focus is on client use of the statistical environment within the database, the embedded statistical environment can also be used directly by the RDBMS server. There is potential to obtain non-trivial performance improvements by using statistical methodologies to control evaluation of queries, including caching, prediction of execution times, etc. We hope this avenue will be explored using the embedded system to facilitate rapid prototyping and experimentation.

2 Prediction

In this example, we assume that the data analyst has explored a training set and decided on a particular model. They have fit that model to the available data and it is now available for prediction purposes. The model may be a parametric model, for example a linear model, or may be a non-parametric fit such as CART. This model can be used within S to make predictions for one or more new records. The user of the database now decides that she wants to predict the value for different records. She does this by issuing an SQL query from her software and is returned a column of predicted values corresponding to the different records identified in her query.

The user will specify the records and variables of interest using a regular SQL command such as

```
select predict(x1,x2) from table
```

This selects all the records in the table and passes each, one at a time, to the predict function. Each call to this function returns, in this case, a single real number. These values are then available to the caller as a column in the result set of the query. A criterion for selecting different records can also be supplied in this SQL expression using the WHERE clause. The details of the invocation of the R function remain unchanged.

How do we arrange to have this query do the appropriate computations? We start with a fitted model computed off-line in an earlier R session. Let’s call this fit $m()$. We store this on disk using the *save()* function in R. Now, within the Postgres server, we retrieve this object and make it available to the R *session*¹

While we have stored fitted model ($m()$) on disk, we could of course have stored it directly within the database. This might be stored as a binary object (blob), using its XML representation, or any other serialization approach.

Next, we register a Postgres function named predict that will forward the request to R. We do this using the CREATE FUNCTION command in Postgres. We specify the name of the function and the types of the arguments. In this case, we specify the argument types for the two columns within the table that we will pass as predictors. These are both real-valued variables so we specify these as float*. We should note that this can be automated from within a client application. For example, using R as a client, we can issue a query to retrieve the types of the columns in the table and then generate the CREATE FUNCTION call. If we want only a subset of the columns, then we will have to identify these and it may be more expeditious to specify the signature (argument types) of the function directly. So, as we build up the SQL command to define the function, we have the following at this point:

¹Currently, there is a single session within the entire Postgres session. However, in the near future there will be multiple interpreters.

```
CREATE FUNCTION predict(float8, float8)
```

We might prefer to have the function definition indicate that one could call it with an entire tuple rather than having to explicitly enumerate each column in the table. This would make it more useful within other tables and also allow the caller to specify different columns directly.²

Next, we specify the return type of the function. Again, this is real value in our example. This adds a little more to the command to to give:

```
CREATE FUNCTION predict(float8, float8) RETURNS float8
```

The next piece of the Postgres function definition involves defining the function itself. In this case, we want to call the *predict()* function, transforming the arguments from Postgres into a data frame with a single row. We pass this data frame and the previously fitted model to predict and have it return the result. To do this, we define a new (anonymous) S function that performs these steps, giving it Postgres as the string value of the AS clause.

And finally, we specify the language in which the function is actually implemented. This identifies where Postgres should dispatch (or “send”) the call. Since this is a record-wise function, we use the procedural language `pl_R`. At this point, we have all the elements of the command to define the Postgres function that will call R to perform the prediction.

```
CREATE FUNCTION predict(float8, float8) RETURNS float8
AS 'function(x1, x2) {
    predict(model, data.frame(x1=x1,x2=x2))
}'
LANGUAGE 'pl_R'
```

Before invoking this function, we must load the *model()* object into the R session. We can do this using the load function that we have defined earlier as part of the `pl_R` Postgres extension language.

```
SELECT load('/home/duncan/model.RData');
```

In the future, we will define this function in a more generic manner so that it takes a tuple with arbitrary variables and constructs the data frame using the names and corresponding values of these variables. In other words, we will have something like

```
function(tuple) {
  predict(model, as.data.frame(tuple))
}
```

The tuple is effectively a named list and so the call to *as.data.frame()* will create a data frame with one row.

Calling *predict()* within the server has allowed us to avoid transferring data from the server to the client and to operate on the data in-place on the powerful server. Additionally, we have been able to use existing software (the prediction function) *without modification*.

2.1 Multiple Models and Closures

The reader might ask how we could handle prediction for two different models. Since the function we defined uses a global variable (*m()*), we would have to ensure that we had assigned the fitted model to this variable before we execute the query. This reliance on global variables is bad software design. While it would be better to pass the fitted model in the query, this is not feasible given the way SQL and Postgres work. Instead, we want each instance of this function to know about its own model. We can do this lexical scoping and closures. We first define a function in R that returns the *predict()* function.

```
predictGen <-
function(model) {
  f <- function(...) {
    predict(model, data.frame(...))
```

²We will work on this

```

    }
    return(f)
}

```

Now, when we call the *predictGen()* function, we get a new instance of the function it returns. Each of these has access to the *model()* object with which the *predictGen()* was called. This allows us to create different prediction functions that can work with different models.

Now, instead of supplying the definition of an R function within the AS clause of the Postgres function definition, we specify an R expression that calls *predictGen()*. We do this as

```

CREATE FUNCTION predict(float8, float8) RETURNS float8
AS 'predictGen(load('model.Rdata'))'
LANGUAGE 'pl_R'

```

(Notice the double quotes within the outer " pair). The AS in this Postgres command calls *predictGen()* with an R object that is loaded from a serialized version of the previously fitted model. We can create a different Postgres function, say *predict1*, in the same way but specifying a different file name in the call to *load()* from which to read the previously fitted model.

3 Record-wise Model Fitting: Regression

In this example, we process multiple records and compute a result that is aggregation of these. These types of *aggregate* functions can be used in statistical computations to fit models, compute statistics for groups of observations. In this example, we use

Many statistical algorithms can be readily computed record-wise. A challenge is to program these and handle others. We do not need to work on individual records but can gather blocks of observations.

Our example will compute the median of a variable. We use a statistical approach to estimate the median rather than store all the values in memory. This is based on work [?] by John Chambers, David James, Dianne Lambert and Scott Vander Wiel and uses the R package developed by David James. The package uses a class of "object" that supports methods for merging new observations with those previously processed and updating the estimate of the quantiles of the variable. The constructor or generator function is named *agentIQ()* and its first method is *merge()* and the second is *median()*³

We define the aggregator function with the following command.

```

CREATE AGGREGATE rquantile(
  basetype=float8,
  initcond1='agentIQ(100)'
  finalfunc=pl_ragg_float8_result,
  stype1=RAggregator,
  sfunc1=r_update_float8,
);

```

The important elements of this command are the first 2, and the others follow closely from this information. We firstly specify that the aggregator works on real numbers ((float8)). This establishes the type of the variable being processed.

Next, we specify how to create the object that is to be updated with each record. This is an R command which should return such an object. In this case, the function *agentIQ()* is called and the buffer size is given as 100. For each collection or group of records, this R expression will be evaluated. Therefore, for each invocation of the *rquantile* function, or for each group of records in a GROUP BY query, we will create a new instance of the updating object. Each of these will act independently of the others.

Since we this aggregator function is dealing with individual float8 values, it is natural to use the *r_update_float8* function as the one to do the record-wise updating. This is a function provided by the embedded R facility and is used for updating an updatable R object with one real value. We make the broad class of updatable R objects known

³This uses a slightly modified version of R package in order to simplify the example.

to R as the Postgres data type RAggregator. This is a general R object, but usually provided in the form of a list of functions defining a closure. It should have at least two methods. The first is the update function which is called for each record in the query with the value(s) in the SELECT clause. The second function is the one that is called to get the result when all the records in the group have been processed. This takes no arguments and is expected to return the aggregated result. In our case, this is the *median()* function.

One need not have these functions as the first and second elements in the list of functions returned by the initialization expression. Alternatively, one can specify them as named elements in the list with the names `update` and `result` respectively.

In our example, we modified the function *agentIQ()* so that the second element in the list of functions it returns was the final function that returned the estimate of the median. The unaltered package has the *quantile()* function as the second element of the list. We cannot use that since we have to specify which quantile we are interested in.

One approach is to append a *result()* function to the list returned by the call to *agentIQ()*. We do this in the `initcond1` attribute in the call to CREATE AGGREGATE. In our example, we would have something like

```
CREATE AGGREGATE rquantile(
  basetype=float8,
  finalfunc=pl_ragg_float8_result,
  stype1=RAggregator,
  sfunc1=r_update_float8,
  initcond1=' tmp <- agentIQ(100)
              f <- function() { quantile(.5)}
              environment(f) <- environment(tmp{\Tt{}}1)
              c(tmp, result= f)'
);
```

Note how we have to change the environment of the new function so that it can see the other functions and variables within the closure. We can of course write a function to hide the details of this merging of a new function into a closure. We might replace the four R expressions with a single expression of the form

```
addFunction(agent(100), result= function() { quantile(.5)})
```

4 Data Collection and Filtering

In many situations, data is gathered dynamically, processed by different applications and added to a table in a database. Observations are collected from devices connected to machines and relayed to the database. This is common in manufacturing and gathering web traffic information such as click-streams.

We consider a simple example in which we verify the values within the tuple being inserted into a table. We check that the value of a particular real-value field in the tuple falls in the appropriate range. We start with a table that has, for simplicity, three fields: `identifier`, `age` and `startDate`. The client application will issue an SQL command such as

```
INSERT INTO table VALUES ('123456', 55, '2001-03-17');
```

to put the triple of an identifier, age and starting date into the table. In this study, we limit the age of the participants to be between 30 and 39. Therefore, in this case, we want to reject the tuple.

To do this, we want to associate a function that gets called each time a tuple is entered into the table. The function might look something like the following:

```
function(tuple) {
  age <- tuple{\Tt{}}"age"
  if(age < 30 || age > 39)
    return(NULL)

  return(tuple)
}
```

This extracts the value of the `age` variable from the tuple that is being inserted and then checks whether the value is in the specified range. If it is not acceptable, we return **NULL** to signal to Postgres that it should abandon the insertion. Otherwise, we return the tuple that was handed to the function and Postgres will proceed to insert the record.

Now that we have the function, we need only arrange for it to be called. We first define it as a trigger function, and then we associate it with insertion events on the table of interest. We define the function as

```
CREATE FUNCTION checkAge() RETURNS OPAQUE
AS ' function(triggerInfo) {
    age <- triggerInfo$tuple["age"]
    if(age < 30 || age > 39)
        return(NULL)

    return(tiggerInfo$tuple)
}' LANGUAGE 'pl_R';
```

The `S` function takes a reference to the trigger information provided by Postgres. Inside in this is the actual tuple that is being inserted into the table. There are a variety of functions that allow us to get and set the names and values of the variables in the tuple, etc. In this case, we just extract the value of the `age` variable and check that this is within the range. If not, we abandon the insertion by returning **NULL**. Otherwise, we just return the tuple value that was given to us.

Finally, we register the trigger with the table. In this case, we want to be notified before the values are actually inserted so that we can veto it. Therefore, we qualify the trigger event with the **BEFORE** keyword.

```
CREATE TRIGGER foo BEFORE INSERT ON table
FOR EACH ROW EXECUTE PROCEDURE checkAge();
```

Now, issuing the insertion “queries”

```
INSERT INTO table VALUES ('123456', 55, '2001-03-17')
INSERT INTO table VALUES ('100056', 32, '2000-12-10')
```

results in the first being rejected and the second accepted. We check this by issuing the query

```
SELECT count(*) from table;
```

before and after the insertions.

Note that in this case, we are not using R’s statistical capabilities in the function. We are merely using it as a convenient, high-level scripting language. Other useful examples of triggers do use the statistical functionality. We might update aggregate statistics (means, correlation matrices, model fits, etc.) about the table. Alternatively, we might performing transformations on the tuple’s elements such as histogram equalization or coordinate registration for images, and so on. Triggers can also be defined for deletions from tables allowing us to perform the same sort of “updates” but as data is removed.

5 Stand-alone Functions

We have discussed using R functions in terms of operations on tables. Record, aggregate and trigger functions each operate on records associated with tables. However, we can call record-oriented Postgres functions directly and independently of a table. For example, we can invoke the `gamma` function provided in the examples shipped with the package via the query

```
SELECT gamma(4);
```

Similarly, we can define other stand-alone, or table independent, functions that can be implemented using R functions. For example, we can define functions to manage the R session and interpreter. These are regular Postgres functions that are members of the `pl_R` language and defined as in section 2.

We can provide a function for attaching and detaching R packages/libraries. This might be defined as

```
CREATE FUNCTION library(text) RETURNS int4
AS 'function(x) {library(x); T}'
LANGUAGE 'pl_R';
```

Similarly, we can provide functions for examining the variables in the global environment using the *objects()* function. In this case, it returns an array of strings – the names of the variables – and is a simple, direct call to objects and so needs no AS clause. The declaration can be given as

```
CREATE FUNCTION objects() RETURNS _text AS '' LANGUAGE 'pl_R';
```

This support for reflectance on the session allows privileged users to *externally* monitor and repair the R interpreter running within the server. This will hopefully allow non-intrusive diagnostic and maintenance actions without having to restart the server.

6 Current Status

We have developed software for embedding the statistical environment R within the Postgres RDBM server, and also the Omegahat and Java interpreters within MySQL. I currently feel that the RDBMS software on which we should focus our efforts are Postgres and Oracle. MySQL was not designed to admit such extensions. While it has been possible to add them, installing them requires modifying the MySQL code. This makes maintaining the extension complex and supporting different variants more time consuming.

This package follows our original work to modify the internals of MySQL (version 3.23.16) to support user defined functions (UDF) implemented in interpreted languages, specifically Java and Omegahat. Because MySQL was not originally designed to be extensible in this manner, and also does not support the rich set of object features that Postgres does, it is likely that we will not pursue the MySQL approach as part of our research. We encourage any interested to contact us and perhaps use the code we have developed. We will probably focus on extending the embedded language approach within both Postgres and Oracle.

7 Future Work

We have not mentioned any details about performance improvements that can be achieved using this embedding approach. This requires careful attention and an appropriate experimental design to account for the numerous factors that will influence the performance of the embedding and the client approaches. The obvious factors include available network bandwidth; the computational resources of the server (CPU(s), RAM, I/O speed); the average number of concurrent queries and the load on the server; the size of the tables being accessed; and so on.

A glaring omission in the current version of the package is the ability to convert R objects to Postgres arrays. This will be added shortly.

Minor enhancements to the code can be made for use with Postgres 7.1 which should realize large performance improvements. The ability to locate the S function just once per query and hence avoid the cost of per-record lookup should be quite significant in performance terms.

We may also explore storing S objects within the database itself, using binary, XML and text representations to serialize the objects. Additionally, we will explore using S objects and classes to exploit Postgres's extensible data types at the user level. (We already use this extensibility in defining the RAggregator type.) As mentioned above, we will also explore how we can pass a tuple as an argument to the record and aggregate handler functions. It is not clear that this is always possible.

We may also add functionality to R so that the R interpreter running within the Postgres server can access the tables within the server. This involves creating an interface between R and the Postgres Server Programming Interface (SPI). The tuple access in the trigger functions already uses this, and it is reasonably straightforward to add explicit support at the S language-level for the entire interface.

There are many enhancements that are needed to the statistical systems before one can deploy the embedded R within the RDBMS server approach in “production” systems. Most RDBMS servers are multi-threaded, while most statistical software is not. We are in the process of making R support multiple interpreters, and then hopefully concurrency/parallelism [3]. We need to add a security infra-structure to R so that users invoking R functions have

limited access to low-level system functions. They should not be able to access data in other concurrent R interpreters within the database. Nor should users be able to load their own C code or execute calls to the underlying operating system (e.g. using *system()*). And we must have a mechanism to identify and prohibit denial-of-service (DoS) attacks caused by consuming the servers resources (CPU cycles, disk space and access, etc.) These are active areas of research for some of us.

References

- [1] The Postgres development team. Postgresql. <http://www.postgresql.org>, valid April 2001.
- [2] The R development team. R. <http://www.r-project.org>, valid April 2001.
- [3] Luke Tierney. Threading and GUI Issues for R. <http://www.stat.umn.edu/~luke/R/thrgui/thrgui.pdf>, March 2001.