

Embedding Foreign Systems in Postgres

Duncan Temple Lang

December 11, 2000

Abstract

We describe a mechanism by which two high-level statistical computing environments are embedded within a Relational Database Management System - Postgres. The existing functions in these systems can be accessed via SQL commands executed in the RDBMS. Additionally, privileged users can define new SQL functions that are implemented in either of these interpreted languages. The benefits of such embedding are

- a that users of databases have access to sophisticated statistical methodology without needing to learn new languages/software, and
- b large performance improvements are possible by avoiding the transmission of a large volume of data from the database to the client, and also the powerful environment typical of database servers can be used to process queries in place, rather than on less powerful client machines.
- c immediate reuse of existing code in a new context.

Contents

1	Aggregate Functions	2
1.1	Example: A Single Quantile - Median	2
1.2	Example: Array Values - Quartiles	3
2	Issues, Questions, etc.	3
3	Generic Implementation	4
4	Class Instances as Arguments: Element Access and Conversion	5
5	Procedural Language & Functions	8
5.1	Example: T -distribution	8
5.2	Example: <code>rnorm()</code>	9
5.3	Example: Γ Function	9
5.4	Procedural Language Scripts	10
6	Performance Considerations	10
7	Security	11
8	Compilation	11
9	Running Postgres	11
9.1	The JVM Environment	11
10	Testing The Procedural Language Interface	11
11	Issues	12

1 Aggregate Functions

Postgres provides support for embedding a Procedural Programming Language (PL) and registering functions within that language as SQL functions.

Fundamentally, statistical operations “reduce” data from observations or records into statistics. The challenge lies in connecting the object-oriented systems in which records are processed by the same instance and the procedural language approach which is state-less. Basically, we want to connect the procedural language (OmegaHat, Java, R) with the aggregate functions. Similarly to MySQL, Postgres provides a way of defining aggregate functions. This is slightly more elaborate in that it provides a type system. We want to be able to connect a closure in R or a Java object (an instance in each language) with a particular call to the function. Postgres allows us to provide an initializing call in the form of a string to a data type.

The goal is to be able to define different aggregate function types. For R, we might have an `RStatelessFunction` and an `RClosureFunction`. These are the same basic internal types in R, a `SEXP`, but are differentiated the Postgres level. The idea is that we create an instance of the appropriate function type and initialize it appropriately. Thus we must be able to provide constructor arguments. Something like

```
[ ]
funcName(arg1, arg2, namedArg1=value, namedArg2=value)
```

should identify the function name and the list of constructor arguments. For a closure, this is an actual function call to generate the instance. For a regular function, we might want to interpret this differently. But, basically, we only want to deal with closures!

To interpret this constructor/initialization string, we use the OmegaHat or R parser.

For the aggregate function to work easily, we need a single C transition routine to be registered for all aggregates. (Otherwise we would have to create a new one for each aggregate function.) The different

```
[ ]
CREATE AGGREGATE name (
    BASETYPE = input_data_type, # record/argument type for each call to
                                # sfunc.
                                # Something int4, Point, javaObject
    SFUNC = java_aggregate_handler,
    STYPE = javaObject,
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ] )
```

1.1 Example: A Single Quantile - Median

Consider the following example. We want to compute the median of a column of data. We use a `Quantile` class and initialize it with the desired quantile “level” - 0.5. We give its constructor some additional arguments such as buffer length, etc. This is achieved in the initial condition clause below.

We indicate which type of data on which this `Quantile` operates. In this example, we specify this as `float`. (Does Postgres convert values to this type like MySQL can?)

The `Quantile` class must have an appropriate method that is called when each record is given to it. Specifically, we must have at least one of the following.

```
[ ]
add(float)
add(float[])
add(Object[])
```

This is exactly the same circumstance as the MySQL embedding. We will have greater efficiency but more work if we support the `add()` for each of the primitive types. However, there are few of these and we should do this and cache the method type along with the Java reference so that we can avoid recomputing it for each record (i.e. call to the state-transition function).

Note that the state transition function is specified as `java`aggregate`handler`. This is the bridge between the Postgres engine and the R or Java engine.

Should we have a Java foreign function object that has a reference to a Java object and also caches information about the method(s) being invoked on that object.

What about Java objects moving, etc.? Make certain we have a global reference. If we get an error, have to arrange to release the reference to the Java object, et al.

```
[ ]
CREATE AGGREGATE median (
    BASETYPE = float,
    SFUNC     = java_aggregate_handler,
    STYPE     = JavaObject,
    FINALFUNC = getDoubleValue,
    INITCOND  = "Quantile(.5, 33)"
)
```

1.2 Example: Array Values - Quartiles

Suppose we want to get back more than a single scalar value. Then, we initialize the *Quantile* object differently and have it compute more than one quantile. The other thing to change is the return type that Postgres expects. This is indicated by the return value of the final function (the `FINALFUNC` clause). The `getDoubleArrayValue` is registered with Postgres giving the return type `float8`, indicating that it is an array of `float8` elements. The `getDoubleArrayValue` function is given the `JavaObject` that represents the *Quantile* object which stores the computed values. This then requests the `double[]` and converts it to the Postgres array.

```
[ ]
CREATE AGGREGATE quartiles (
    BASETYPE = float,
    SFUNC     = java_aggregate_handler,
    STYPE     = JavaObject,
    FINALFUNC = getDoubleArrayValue,
    INITCOND  = "new Quantiles(new double[] {.25, .75}, 33)"
)
```

The simplest way for us to implement the generic aggregate handler is to introduce two new procedural language definitions. These handle the state-transition function calls and the final function wrapup functions. In Java, the first of these converts its arguments to obtain the *PostgresUDF* object and call its `add()` method with the value of the second argument.

The wrapup function is similar, but calls the `result()`.

2 Issues, Questions, etc.

Can we have multiple variables within a single call, or do we have a single object/instance that contains these values? (The latter wouldn't be general enough as we may want an expression such as the second one below)

```
[ ]
SELECT cor(x,y) from table;

SELECT cor(x,y) from table where X table1.A, Y table2.B;
```

How are these passed to the state-transition function? As a single object or as multiple arguments? It appears from the aggregate specification that there can be only a single argument.

Can we perform a two step operation by selecting them into a new instance of an object (an array) and then passing this to the aggregate function.

```
[ ]

SELECT cor( newFloatArray(x,y) ) from table where X table1.A, Y table2.B;
```

Since this is a very generic interface between the two systems (Postgres and R or Java), we want the internals of the receiving system (R or Java) to be able to determine the types the host system expects both for input and input arguments. Since Postgres is catalog based, we can query these values from within the C code that implements the bridge between the systems.

While we typically use regular full-fledged Java objects, it is convenient to use primitive types. Consider the simple scalar aggregate functions such as sum, mean, max, min, etc. In these cases, we want the transition type to be a real number (double or float). Similarly, we have no apparent need for a final or wrapup function. How do we conveniently handle these? Well, we don't need state, but we do need a simple R function or a Java object. Consider the case of R and the sum function. We might register a rsum via the R procedural language interface. The R function might be something like

```
[ ]
function(old, val) {
    old + val
}
```

The state-transition type here is an R function. Thus we need a final function to indicate that the return type is a scalar value, not the function itself. We use the `getDoubleValue` which in this case turns the single float into a float via the identity function. We can thus supply the basic identity functions for this purpose.

What about examining the name of the function being called and handing that to the foreign engine, as we do in MySQL when registering functions. This can be done since we get the name of the function via the

We can define a collection functions that convert from a foreign object to a Postgres value. This collection contains entries for converting a Java object to an integer, float, double, long, logical/boolean, CORBA object and general R object.

3 Generic Implementation

We mix the Types and Aggregates to implement this. We define the *state type* as the Postgres type `JavaAggregator`. This is a simple class that has an input method that is an OmegaHat expression which constructs a Java object. The Java object must implement the `PostgresUDF` interface. The *initial condition* argument is this constructor call to create such an `PostgresUDF` object. This is different for each aggregate function. This might be something like

```
[ ]
"new Quantile(.5)"
"new Quantile(new double[] {.25, .75})"
```

The `JavaAggregator` class stores a reference to this object. This might be an Omegahat name by which the object can be retrieved, or the value of *C*-level pointer which can be cast to a *jobject* in subsequent *C* routines.

The value of the *sfunc* argument is then given as the generic `javaAggregatorHandler`. This is given the value of the current record and also the `JavaAggregator` object. It uses the later to invoke a JNI call on the associated `PostgresUDF`, passing it the record value, having converted it appropriately.

The *ffunc* cannot be entirely generic. Instead, it must be a function whose return type makes sense for the aggregate function. That is it must return the desired type such as `float8`, `int4`, or array versions of any of these (e.g. `int4`). We provide *C* routines that do this for the basic types.

The State-transition function and the final function might be implementable using the Omegahat Procedural Language - **plomegahat**.

4 Class Instances as Arguments: Element Access and Conversion

Consider the canonical example used in the Postgres documentation of an employee class `EMP`. Suppose we have a collection of instances of such a class and we want to pass each one to a foreign system's function. We have a composite Postgres type and we want to operate on it within a foreign system which does not directly understand the layout, etc. of Postgres types.

A simple mechanism for transferring objects is to use the approach in the Tcl procedural language interface. This translates the Postgres object to an associative array whose elements are indexed by the field names of the Postgres instance and the elements are the values of those fields.

This approach is natural in Java/Omegahat and R. In R, we would use a list to store the name-value pairs. The function being called could elect to dereference the elements explicitly (via the `$` or `[[` operators) or have the expression(s) be evaluated with the list treated as an evaluation frame/environment. In the Omegahat, the approach is akin to creating an evaluation frame/database (implemented using an `OrderedTable`). However, this has the potential to be very expensive. This is true from the perspective of both setting and retrieving the values.

An alternative for the Java/Omegahat interface is to provide classes that correspond to the Postgres class. For example, consider the `EMPLOYEE` defined in the Perl procedural language section (sect 12) of the User's Guide[?].

```
[ ]
CREATE TYPE EMPLOYEE (
    name text,
    basesalary int4,
    bonus int4 );
```

We can create a Java class `Employee` that mirrors this, containing three corresponding fields and providing accessors. When we convert from the Postgres instance of this class to Java, we call the appropriate constructor with values for the different fields.

When such a composite object has fields which are also composite types, the conversion is slightly more involved and must be nested. For example, consider the two class definitions below. These define an employee as part of a department. (We would probably want to have a key to a different table rather than have the department in the object itself, but that raises another point on this topic!)

```
[ ]
CREATE TYPE DEPARTMENT (
    head text,
    numEmployees int4,
    orgCode int4
)

CREATE TYPE EMPLOYEE (
    name text,
    basesalary int4,
    bonus int4,
```

```

    dept department
)

```

The Java/Omegahat command to create an instance of Employee would then be

```

[]
new Employee(name, basesalaray, bonus,
             new Department(head, numEmployees, orgCode))

```

corresponding to the constructors in the two classes derived from the Postgres class definitions.

```

[Java]
Employee(String name, int baseSalaray, int bonus,
         Department dept)

```

```

Department(String head, int numEmployees, int orgCode)

```

Note that the Omegahat facilities allow us to (relatively easily) dynamically generate these derived classes. By reading the Postgres catalogs in which the types are defined, we can compile Java byte-code that defines the classes in terms of their fields and field accessors.

We can also not guarantee uniqueness that results from pass-by-reference. For example, if two instances share the same instance as a field value, we would have two separate copies on the Java side. The linked list is a natural example of where this fails.

Does anyone care?

To deal with unnecessary copying and maintaining the semantics in the language in which the object was originally defined and created (e.g. as pass-by-reference and uniqueness), we can create a Java class (`EmployeeReference`) which is a reference to the Postgres object. This object would have the same field accessor methods (e.g. `String getName()`, `void setBonus(int)`). However, it would not have fields which stored these values, but instead would maintain a reference to its C-level counterpart that is part of the Postgres system. These field accessors would invoke native methods to return to C code to dereference the appropriate field or set the field's value in the Postgres instance.

The idea is reasonably simple. We would create an instance of the Java `EmployeeReference` class by passing it the address of a Postgres object which represents the `EMPLOYEE` instance. This is given as a `long` in Java and is the only argument in the constructor invocation of `EmployeeReference`.

```

[C]
jobject foo(TupleTableSlot *t) {
    jclass klass = VMENV FindClass("EmployeeReference");
    jmethodID mid = VMENV GetMethodID(klass, "<init>", "(J)V");
    ref = VMENV NewObject(klass, mid, (jlong) t);
    ref = VMENV NewGlobalRef(ref);
}

```

At this point, the `EmployeeReference` has access to the `TupleTableSlot` for as long as it exists. This Java code that implements the UDF can call the field accessors of this `EmployeeReference`. These are implemented in the following manner.

```

[]
public String getName() {
    return(getAttributeValue("name", "java.lang.String"));
}

public Object getAttributeValue(String id, String type) {

```

```

    return(getAttribute(id, type, getAddress()));
}

native Object getAttributeValue(String attrName, String type, long address);

```

The implementation of the native method is something similar to the following (written in C#):

```

[]
jobject PosgresReference_getAttributeValue(JNIEnv *env, jobject theThis,
                                           jstring id, jstring type, jlong address)
{
    TupleTableSlot *t = (TupleTableSlot *) address;
    jboolean isCopy;
    bool isNull;

    const char *idString = VMENV GetStringUTFChars(id, &isCopy);
    const char *value = (char *) GetAttributeByName(t, idString, &isNull);

    if(isCopy)
        VMENV ReleaseStringUTFChars(id, idString);

    jobject val = NULL;
    if(isNull == false || value == NULL || value[0] == NULL)
        val = VMENV NewStringUTF(value);

    return(val);
}

```

Note that the `EmployeeReference` would be defined as extending a basic class for this inter-system interface. Specifically, the storage of the address of the `TupleTableSlot` would be inherited from a class `PostgresReference`. Similarly, the native `getAttributeValue()` is also inherited by such derived reference classes from the base `PostgresReference`. All the effort can be expended into creating converters in that one class.

More information can be provided to the native `getAttributeValue()` method to help it in casting the value returned by `GetAttributeByName()` appropriately. This is know when the class is defined based on the definition of the Postgres “class”. This type can be passed as a string or enumerated type. Then the `getAttributeValue()` can be written more generically as

```

[]
jobject PosgresReference_getAttributeValue(JNIEnv *env, jobject theThis, jstring id,
                                           jstring targetType, jstring fromType, jlong address)
{
    TupleTableSlot *t = (TupleTableSlot *) address;
    jboolean isCopy;
    bool isNull;

    const char *idString = VMENV GetStringUTFChars(id, &isCopy);
    const char *value = GetAttributeByName(t, idString, &isNull);

    if(isCopy)
        VMENV ReleaseStringUTFChars(id, idString);

    if(isNull)
        return(NULL);
}

```

```

    jobject val = convertPostgresToJava(value, targetType, fromType);

    return(val);
}

```

The `convertPostgresToJava()` would then perform the appropriate casting and conversion.

Note that an enumerated type would not be extensible for handling composite types that are introduced into the system after the native method has been compiled. Accordingly, strings are used so that they can explicitly reference the Java and Postgres classes being converted to and from.

This setup is similar to the style of (primitive) object/value conversion as used in the CORBA and Java packages for R are needed and would work here.

Again, such a class can be easily generated automatically and dynamically via the Omegahat byte-code generation facilities.

Input and output functions. Convert to strings and then to Java or R types from these and then back again.

5 Procedural Language & Functions

There are two types of functions to call in Java - static and non-static. Static methods are simple. We can treat these as simple Omegahat functions and invoke them. We can use the local function mechanism by which a class or object can be attached to the search path of the evaluator and the functions within that class or instance can be called without the qualification of a target class or instance. For example, the following commands are in Omegahat and Postgres respectively. (The `max` function in Postgres would be used instead of the Omegahat one.)

```

[Omegahat]
addFunctionTable(Math)

```

```

[Postgres]
CREATE FUNCTION max(float8, float8) RETURNS float8
  AS 'max'
  LANGUAGE 'plOmegahat';

max(a, b)

```

5.1 Example: T -distribution

Consider the following simple and contrived example. Suppose we have a field or column (**df**) in a table that contains a collection of integer values. These are to be used as the degrees of freedom of a T distribution. We can use R to generate deviates from the different T distributions, parameterized by those degrees of freedom parameters.

```

[]
CREATE FUNCTION  rt(int4) RETURNS float8
  AS ' as.numeric(rt(1, $1)) '
  LANGUAGE 'plr';

```

```

[]
SELECT  rt(df) from table;

```

5.2 Example: `rnorm()`

Instead of defining a new function within Postgres, we can use existing R functions. We can treat the values in the `df` field as the number of deviates to generate from a standardized normal.

```
[ ]
CREATE FUNCTION rnorm(float8) RETURNS _float8
AS '' LANGUAGE 'plr';

SELECT rnorm(df) from table;
```

5.3 Example: Γ Function

As an other example, let's use a mathematical function provided by R that is not directly available in Postgres: the gamma function. We want the user to be able to call this with a numeric value and have the result of the gamma function be returned.

```
[ ]

SELECT gamma(df) from table;
```

To do this, we register the gamma function as a function in the procedural language R. We indicate that it is a "built-in" function of R by giving it no body.

```
[ ]
CREATE FUNCTION gamma(float8) RETURNS float8
AS '' LANGUAGE 'plr';
```

Similarly, we can make any of the other mathematical functions in R available to Postgres expressions.

To register a PL, we first define and register (with the Postgres engine) a C routine that acts as a bridge between Postgres and the PL. Then we register the language with Postgres and connect it to the handler routine. We are now ready to define and register functions in the PL.

```
[ ]
CREATE FUNCTION plomegahat_call_handler () RETURNS OPAQUE AS
'/usr/local/pgsql/lib/plOmegahat.so' LANGUAGE 'C';

CREATE TRUSTED PROCEDURAL LANGUAGE 'plomegahat'
HANDLER plOmegahat_call_handler
LANCOMPILER 'PL/Omegahat';
```

The C routine that handles the calls to functions in a procedural language has the following signature.

```
[ ]
Datum handler(FmgrInfo *proinfo, FmgrValues *proargs, bool *isNull);
```

The *FmgrInfo* and *FmgrValues* are defined in the header file `fmgr.h` in the `src/backend/` directory of the Postgres distribution. (This one is identical to the one in `src/backend/utils/` apparently.) The number of arguments the function expects is given by the field `fn_args` and the actual arguments are in the *FmgrValues* object's data array field. *TupleTableSlot* pointers.

```
[ ]
TupleTableSlot *t = (TupleTableSlot *) proargs->data[i]
```

The types of the arguments and the return value are available by querying one of the catalogs maintained by Postgres. The information about each argument and value is accessible from an object of type *Form_pg_type* defined in **pg_type.h** in the **include/catalog/** directory.

5.4 Procedural Language Scripts

It is often non-trivial to specify a function in the procedural language (Omegahat or R) ahead of time so that it can be called from a Postgres SQL command. There are at least two occasions where this is the case. The first is where we want to call a function in R or a top-level method in Omegahat whose name is also an existing function name in Postgres. For example, there is a Postgres function `pow` and also the static Java method `pow()` in the `Math` class. We need to be able to invoke both, unambiguously and without conflict and relying on different signatures.

The second occasion where we cannot simply associate a “method” in the procedural language with the same name in Postgres is when we wish to dynamically declare a new function. This is possible in a procedural language in Postgres if it is trusted. The facility allows users to specify the expressions to be executed in the procedural language via the `AS` clause of the `CREATE FUNCTION` command. This essentially defines the body of a function. This makes the procedural language and Postgres extensible by (privileged) users.

To support these scenarios, the Omegahat procedural language handler supports expressions in the `AS` clauses of the `CREATE FUNCTION` command. If the function being called has a non-empty body, the contents are passed to Omegahat and are evaluated each time the function is called.

The function body would be relatively uninteresting if it did not operate on the values of the record on which it is called. In other words, we must be able to write the function body in such a way that it can access the arguments to the function call and operate on those. We use the convention that the arguments are named `$0`, `$1`, ...

Take the example of `pow()` above. To be able to call this Java method as a Postgres function, we might create a the Postgres function as follows

```
[ ]
CREATE FUNCTION jpow(float8, float8) RETURNS float8
AS 'Math.pow($0, $1)'
LANGUAGE 'plomegahat';
```

A simple function that raises the argument to the power of 2.5 is given below.

```
[ ]
CREATE FUNCTION jmisc(float8) RETURNS float8 AS 'Math.pow($0, 2.5);' LANGUAGE 'plomegahat'

select jmisc(3);
```

We can use a trigger to be notified when the PL function is dropped. This allows us to discard the cached version of the function.

Derivatives or different dialects of the basic Omegahat PL can be registered which process their arguments differently. For example, we might have a static method call where the name of the Postgres function was the name of the method and the `AS` clause contained the name of the class in which the method should be called. Another dialect may use the `AS` clause to specify a properties table of name-value pairs which would be used to .

6 Performance Considerations

Look at the `CREATE LANGUAGE` for information about caching.

7 Security

First of all we must remove insecure functions such as *system()*, and limit others such as *cat()*, *write()*, etc. that access the file system. (Connections would be a welcome centralization here.)

We should note that the RSPerl and RSPython packages could prove to be quite useful in this context. R makes use of the shell by dispatching commands to it and gathering the output. For example, it calls *wget*, *lynx*, etc. Many of these operations can be done directly via the *.Perl()* interface and do not raise security issues, assuming that the Perl interpreter is also secure.

8 Compilation

It is *vital* that the Postgres distribution be compiled with the pthread library if the different JVM implementations are to be correctly loaded and initialized. A “simple” way to get this to work is to edit the **Makefile.global.in** file in the **src/** directory of the Postgres distribution. The values are propagated to the different executables. Perhaps a more refined version of this is just when compiling the **postgres** executable.

There are also problems with instantiating different classes (e.g. *MySQLOmega*).

9 Running Postgres

To be able to load the shared library that provides the Omegahat and specifically JVM facilities, one must have the correct value for the `LD_LIBRARY_PATH` environment variable for the backend **postmaster** process.

For the IBM JDK, the following is appropriate

```
[ ]
${OMEGA_HOME}/DBMS:${OMEGA_HOME}/lib:/usr/local/src/IBMJava2-13/jre/bin:/usr/local/src/IBM
13/jre/bin/classic
```

The `$OMEGA_HOME/lib` entry provides the *libRSNativeJava* and the *libProps*.

9.1 The JVM Environment

Clearly, one needs to be able to control many of the run-time characteristics of the Java Virtual Machine. For example, one will want to be able to provide a suitable classpath. Similarly, one may wish to specify different security settings, initial heap and stack sizes, what class to use as the default evaluator, etc.

The code that loads and initializes the Java Virtual Machine is configured to read properties - name: value pairs - from a file. These are read by the C code that initializes the JVM and also the Omegahat class that control the evaluation of the Omegahat functions, scripts and the aggregate functions.

The name of the properties file is specified by the environment variable `MYSQL_JVM_PROPS`. The C code treats some of the properties specially, expanding them so as to be used by the JVM initialization. The primary one of these is `System`. These values are appended to the options passed in the creation of the JVM. These are specified as elements in a space-separated list of `-Dname=value` entries.

10 Testing The Procedural Language Interface

The `Math` class is attached by default to the default evaluator created by the embedded Omegahat session in the form of an instance of *PostgresOmegahatManager*. Thus, we can invoke its static methods as if they are Omegahat functions. For example, we can call its `rint()` method.

```
[ ]
CREATE FUNCTION rint(float8) RETURNS float8 AS '' LANGUAGE 'plomegahat';
```

We can the invoke this function with a Postgres command such as

```
[ ]
select rint(3.4);
select rint(3);
```

11 Issues

We currently have to wrap the arguments in a *ConstantExpression*.

The Postgres engine cannot unload the shared library as we cannot restart the JVM. Need a flag for this in the Postgres catalogs, or a special routine or symbol in that library that indicates whether unloading is ok.

The Java code should write to `elog`, not `stderr`.