Enhancing PL/pgSQL to support packages

PL/pgSQL is an imperative procedural programming language. Many such languages support constructs to group related functions and procedures into a single containing unit. (I'll hereinafter use the term *subprogram* to mean either function or procedure. It's unimportant that some languages don't even use the keyword *procedure*.) For example, Python lets you group a main program and the helper subprograms that it uses within one single file so the these units can refer to each other without qualified names. It also lets you group generic helper subprograms into one file so that you can then use these helpers from subprograms grouped in other files, using suitably qualified names.

I'll use the term *module* in this essay as a generic term of art for the grouping construct in any language that supports such a construct. A *module*, across a range of languages, usually supports the declaration of so-called *module-global variables* that are usable from all of the contained subprograms but (maybe according to programmer choice) are inaccessible from outside of the *module*. Moreover, such *module-global variables* hold their state when the point of execution leaves the *module* and then, later, comes back to it. (This behavior is notably different from how the local variables that are declared within a subprogram behave.) Another common notion allows just an identified subset of the *module*'s subprograms to be externally visible, and thereby to define its API, while the other contained subprograms remain firmly hidden behind the API. A *module* has a name, a purpose, and is the basic unit of code management, version control, and deployment.

A "library" notion goes hand-in-hand with a language's *module* construct. And application programmer productivity, in many languages, is hugely enhanced by library *modules* that ship with the language system—each for a particular generic purpose. Additionally, *modules* for the language in question are often available from third parties.

But PL/pgSQL has no such *module* language construct. All it has is individually created functions and procedures. It is possible to establish a set of related PL/pgSQL subprograms with a single dedicated user and/or in a single dedicated schema.

(It is possible to create a PostgreSQL extension that exposes several PL/pgSQL subprograms. They might be implemented in PL/pgSQL, or in another language like C. But if PL/pgSQL is the implementation language, then extension developers feel the absence of a *module* construct in just the same way that they do when using no more than *create function* and *create procedure*.)

PL/pgSQL functions and procedures can be compared to their PL/SQL counterparts in Oracle Database. They're similar in syntax and semantics—and especially in that they support SQL statements as a particular kind of statement within the superset of the programming language's other statements, and that they execute within the same database session that executes SQL. But PL/SQL supports the *module* construct that PL/pgSQL lacks. It's called a *package*—comprised of a *package specification* (hereinafter the *spec*) and a *package body* (hereinafter the *body*).

In this essay, I take a simple use case that can be implemented both in Oracle Database and in PostgreSQL. I show how its two user-facing subprograms can be implemented in a single PL/SQL package which hides a couple of helpers in the body. It also uses package state to advantage. And I show how the code would look if it were mapped to PL/pgSQL syntax using an intuitive extension to

the way that a PL/SQL function of procedure is mapped to its PL/pgSQL counterpart. Then I show how you must implement it today in the absence of a PL/pgSQL *module* construct. I call out the various ways in which this implementation suffers because of that lack.

Why use PL/pgSQL stored procedures at all?

Here are three points on the spectrum of possible answers.

- *Just say "no". Never implement user-defined code that executes inside the database.*

  I've certainly heard this viewpoint vigorously expressed. I'll take the fact that PL/pgSQL is supported at all as reason enough not to debate these nay-sayers.

- *Use PL/pgSQL functions sparingly to implement what might have been SQL built-in functions.*

  Greatest common factor is a good example. (I do realize that PG 13 brought the *gcd()* built-in. But this only strengthens the usefulness of the example.)

- Use PL/pgSQL subprograms to define the entire API, for database clients, of the overall application's use of the database.

  This approach relies on a designed set of users where some own schema objects and one, or a few, others own no objects but can use the subprograms that define the API. The approach ensures that database clients aren't able even to discover the names of the tables, table columns, indexes, constraints, and so on that lie behind the API. This implies that clients mustn't be able to read the code of the PL/pgSQL implementations. This end-point of the spectrum represents the full-on adherence to the principles of modular software design. It regards the database as one component in the modular decomposition of the overall multitier application. Tautologically, the database component must expose its functionality via a business-purpose-oriented API and must hide all the implementation details behind that API. I'll call this the *hard shell* approach just to save words later on.

I realize that some designers are vigorous advocates of the hard shell approach and that others vigorously oppose it. This essay's purpose is met simply by saying that PostgreSQL must allow those who favor the hard shell approach to implement it successfully.

The use case that the code in this essay implements

I've chosen a simple, but fairly realistic use case. Suppose that you want to execute one or several database calls from a client in a single session, and that you want to record the total elapsed time. Suppose that you don't want to change the code of what you call to add timing instrumentation. (Never mind how feasible such intervention might be.) If your client is, for example, a Python program that makes all of the database calls of interest, then you can easily read a clock, record what it says, read it again later, and do your arithmetic—all in python code.

So, for the sake of this use case, imagine that the client is *psql* (or, with Oracle Database, *SQL*Plus*). Think, for example, of a *psql* script that calls child scripts and that executes a mixture of DDL statements, DML statements, and statements of other kinds. You decide that you want to implement the timer with a parameterless procedure to reset and start a stopwatch and a parameterless function to read it:

```
procedure: stopwatch.start();
function:  stopwatch.reading();
```

It's implicit that *start()* needs to note the wall clock time at the moment it's invoked, using a suitable memo, and that *reading()* needs to note the wall clock time, again, at the moment it's invoked and then to read the start time from the memo to allow the elapsed time to be computed by subtraction.

In Oracle Database, *stopwatch* is the name of a package for which you create a *public synonym*. In PostgreSQL, *stopwatch* is the name of a schema that you put on the *path* of a user that needs to use it.

Regard the use of a *path* in PostgreSQL and of a *public synonym* in Oracle Database as devices that make the syntax less verbose when you start and stop the stopwatch and that thereby emphasize the similarity of the approaches in the different environments as they appear to the end user. Opinions vary about the advisability of this approach and some developers prefer always to use fully qualified names. That debate is irrelevant for this essay's purpose.

A note about the code

The PL/SQL implementation for Oracle Database and the currently runnable PL/pgSQL implementation for PostgreSQL are included with this essay as tested, re-runnable scripts—to be executed, respectively, using *SQL*Plus* and *psql*. They produce exactly the same results, to a spool file, in the two environments (within the limits of the expected stochastic variability of time measurements).

You can read the scripts to see the details of the implementations. However, these details aren't the point. Rather, the point is the *structure*: how the code is organized. This essay, therefore, shows the structure but elides the low-level code details.

The runnable PL/pgSQL implementation follows this advice:

> *Instead of packages, use schemas to organize your functions into groups.*

from the PostgreSQL documentation section 43.13. Porting from Oracle PL/SQL. However, it doesn't follow the second part of the advice:

> *Since there are no packages, there are no package-level variables either. This is somewhat annoying. You can keep per-session state in temporary tables instead.*

A PostgreSQL temporary table's metadata has only session duration and therefore it must be created afresh in every newly-started session. But there is no way to make a database trigger fire on session creation and so client-side session-startup code must be used. This is uncomfortable because it delegates the responsibility for the proper behavior of the within-database stopwatch to the client.

Instead, the runnable implementation uses the entity that you get by executing this SQL statement:

```
set stopwatch.start_moment to <the text representation of a number>;
```

I'll use the term *user-defined run-time parameter* for this entity. I made up the names *stopwatch* and *start_moment* and followed what seems to be a recommended practice of introducing the informal namespace *stopwatch* so that I can then invent names of "variables" within it, like *start_moment*, without risking collision with names of "variables" in other informal namespaces. This approach has the drawback that only *text* values can be stored this way and so, in the present use case, values must be typecast on the way in and again on the way out.

Of course, my sketch of a straw-man PL/pgSQL package implementation cannot be run. But the low-level code details would be essentially identical to their counterparts in the runnable implementation that uses free-standing subprograms. They would differ only in how the session-global state is accessed. The strawman PL/pgSQL package implementation replaces the use of the user-defined run-time parameter with the strawman's package body global variable.

The helper function *fmt(n in …, template in …) return …*

The ellipses stand for data types that are functionally equivalent in Oracle Database and PostgreSQL but that have different names. In Oracle Database, *n* is *number*; and *template* and the *return* value are both *varchar2*. In PostgreSQL, *n* is *numeric*; and *template* and the *return* value are both *text*. As it happens, the implementation (just a single statement), is spelled identically in both environments:

```
return ltrim(to_char(n, template));
```

The pedagogy here is that this trivial encapsulation is invoked at ten different sites in the function that uses it, thereby making that code noticeably more compact and therefore easier to proof read. Yet it's impossible to claim that *fmt()* might be generically useful. (At least, take it as standing for such a function.) So it calls out to be hidden from general sight and to be impossible to invoke except where it's needed. In Oracle Database, because it happens to be used only by a single subprogram, the common practice is to write it as an inner function in the declaration section of the caller. It would be nice if PL/pgSQL, too, supported inner subprograms.

Arguably, the case for supporting inner subprograms is separable from the case for supporting packages. As it happens, the structure of a package *body* in PL/SQL is identical to that of a parameterless procedure: each has a declaration section that accommodates both variables and subprograms. For this reason, my strawman sketch of the PL/pgSQL package implementation has *fmt()* as an inner function, too. In both the runnable PL/SQL implementation and the strawman PL/pgSQL package implementation, *fmt()* is defined as an inner function in *duration_as_text()*'s declaration section.

The helper function *duration_as_text(t in …) return …*

This is a classic pretty-print function. It returns the input duration with sensible precision and units according to its size. Even though the scheme is well known in many contexts (for example for file sizes in *bytes*, *KB*, *MB*, *GB*, and so on), the particular design here is specific to the present use case and deserves, therefore, to be hidden from general sight.

In Oracle Database, *n* is *number*; and the *return* value is *varchar2*. In PostgreSQL, *n* is *numeric*; and the *return* value is *text*. The implementation is identical in both environments, and the transliteration from one to the other could be done mechanically. Standalone tests show that the PL/SQL implementation and the PL/pgSQL implementation produce identical results over an appropriately wide range of input values. (To test the PL/SQL implementation you must, temporarily, expose the function in the *spec*.)

Implementing the stopwatch in Oracle Database using a package

Here is the DDL that creates the *spec:*

```
create package stopwatch_owner.stopwatch
  authid definer
is
  -- START is a reserved word.
  procedure  start_;
  function   reading return varchar2;
end stopwatch;
```

And here is the elided DDL that creates the *body:*

```
create package body stopwatch_owner.stopwatch
is
  start_moment double precision not null := 0;

  function duration_as_text(t in number) return varchar2
  is
    ...some useful constants...
    -- RESULT is a reserved word.
    result_ varchar2(32767) not null := '?';

    function fmt(n in number, template in varchar2) return varchar2 is
    begin
      return ltrim(to_char(n, template));
    end;
  begin
    case
      when t < confidence_limit then
        result_ := 'less than ~20 ms';

      when ...
    end case;
    return result_;
  end duration_as_text;

  procedure start_
  is
  begin
    start_moment := dbms_utility.get_time()/100.0;
  end start_;

  function reading return varchar2 is
  begin
    return duration_as_text(dbms_utility.get_time()/100.0 - start_moment);
  end reading;

begin
  -- Optional package initialization code.
  null;
end stopwatch;
```

Finally users other than *stopwatch_owner* will need the *execute* privilege on it, like this:

```
grant execute on stopwatch_owner.stopwatch to client
```

If you say *on package stopwatch_owner.stopwatch*, then you get a syntax error. (Oracle Database and PostgreSQL differ here.) But the effect is to allow the grantee to access the elements that the *spec* exposes: its subprograms, variables, and the ability to read the text that defines it—along with Oracle-specific things like types and exceptions that the *spec* declares. Not even the owner can reference elements that are declared in the *body* from outside of the *body*. But it *can* read the text that defines the *body*. Users other than the owner can *never* read the text that defines the *body*. This is regarded as a *critically advantageous* feature of the hard shell approach. Famously, hackers who have seen the code that implements something can work out ways to do evil that they could not manage to do without seeing the code.

The encapsulation brought by the *body* notion therefore guarantees that you can reason about the state of *body*-global variables like *start_moment* just by reading code within an immediately visibly apparent scope.

Further, the helper function *fmt()* is declared as an inner subprogram within the *body*-private helper function *duration_as_text()*. It's invoked at ten sites there, but isn't needed anywhere else in the *body*.

Establishing it as an inner subprogram tersely tells code readers exactly within what scope they need to understand its use—and therefore the scope within which any code changes would need to be made if *fmt()*'s API or meaning is changed.

Notice that *dbms_utility.get_time()* ships with Oracle Database. Though it's in fact a function in a package, it's morally equivalent to a built-in SQL function. It returns centiseconds since the database instance was started. Strangely, Oracle Database doesn't support *extract(epoch from…)*. (Stackoverflow has indignant questions about this; and you can see workarounds that extract various units like *minutes* and *seconds*, then to multiply these values appropriately and add them all up. Crazy, eh?)

Implementing the stopwatch in PL/pgSQL using a package (strawman)

I didn't spend much time on the language design here. I simply transliterated the PL/SQL DDLs to create the *spec* and the *body* into a strawman for their possible corresponding PL/pgSQL DDLs using the spirit of the same general rules as you use, today, to transliterate the PL/SQL DDLs to create a free-standing function and a free-standing procedure to their actual corresponding PL/pgSQL DDLs.

Here is the strawman DDL to create the *spec:*

```
create package stopwatch
  security definer
  language plpgsql
as $spec$
declare
  procedure  start();
  function   reading() returns text volatile;
end;
$spec$;
```

The items that the *spec* shown here declares are known in PL/SQL as *subprogram (forward) declarations*. You can include a subprogram declaration in any PL/SQL declaration section where the partner *subprogram definitions* are found. Some programmers like to write a list of subprogram definitions in the body for all of the subprograms that aren't declared in the *spec*—simply as a documentation device.

Here is the strawman elided DDL to create the *body:*

```
create package body stopwatch
as $body$
declare
  start_moment numeric not null := 0.0;

  function duration_as_text(t in numeric) returns text stable
  as
  declare
    ...some useful constants...
    result text not null := '';

    function fmt(n in numeric, template in text) returns text stable
    as
    begin
      return ltrim(to_char(n, template));
    end;
  begin
    case
      when t < confidence_limit then
        result := 'less than ~20 ms';

      when ...
    end case;
    return result;
  end;

  procedure start()
  as
  begin
    start_moment := extract(epoch from clock_timestamp());
  end;

  function reading() returns text volatile
  as
  begin
    return duration_as_text( extract(epoch from clock_timestamp())::numeric - start_moment );
  end;

begin
  -- Optional package initialization code.
  null;
end;
$body$
```

Notice that properties like *security* and *language* are the same for the *spec* and the *body* so they aren't repeated in the body's code. The same thinking applies for properties of the subprograms that the *spec* exposes. However, the parameter lists must be explicit in both the *spec* and the *body* so that overload definitions in the *body* can be tied to their declarations in the *spec*.

Presumably, the *grant execute* statement would be spelled like this:

```
grant execute on package stopwatch_owner.stopwatch to client
```

Implementing the stopwatch in PostgreSQL using schemas

I created a dedicated user called *stopwatch_owner* to own the schema *stopwatch* to emulate the *spec* notion, the schema *stopwatch_body* to emulate the *body* notion, and six free-standing subprograms, appropriately housed within one or the other of these schemas, to emulate the elements within the *spec* and the *body*. Here are the elided DDLs to create the *body* emulation. Counter-intuitively, they come first in the thinking and the installation flow:

```
create function stopwatch_body.fmt(n in numeric, template in text) returns text stable
  security definer
  language plpgsql
as $body$
begin
  return ltrim(to_char(n, template));
end;
$body$;

create function stopwatch_body.duration_as_text(t in numeric) returns text stable
  security definer
  language plpgsql
as $body$
declare
    ...some useful constants...
  result text not null := '';
begin
  case
    when t < confidence_limit then
      result := 'less than ~20 ms';

    when ...
  end case;
  return result;
end;
$body$;

create procedure stopwatch_body.start()
  security definer
  language plpgsql
as $body$
declare
  start_moment constant text not null := extract(epoch from clock_timestamp())::text;
begin
  execute format('set stopwatch.start_moment to %L', start_moment);
end;
$body$;

create function stopwatch_body.reading() returns text volatile
  security definer
  language plpgsql
as $body$
declare
  start_moment  constant double precision not null := current_setting('stopwatch.start_moment');
  curr_moment   constant double precision not null := extract(epoch from clock_timestamp());
begin
  return stopwatch_body.duration_as_text((curr_moment - start_moment)::numeric);
end;
$body$;
```

Here are the elided DDLs to create the *spec* emulation.

```
create procedure stopwatch.start()
  security definer
  language plpgsql
as $body$
begin
  call stopwatch_body.start();
end;
$body$;

create function stopwatch.reading()
  returns text
  volatile
  security definer
  language plpgsql
as $body$
begin
  return stopwatch_body.reading();
end;
$body$;
```

Each subprogram in the *stopwatch* schema is just a trivial jacket for its partner in the *stopwatch_body* schema. A *grant usage* statement is needed for the *stopwatch* schema, thus:

```
grant usage on schema stopwatch to client
```

And a *grant execute* statement is needed for each subprogram, thus:

```
grant execute on procedure stopwatch.start() to client
```

and thus:

```
grant execute on function stopwatch.reading() to client
```

The design is necessary to achieve the outcome that grantees like *client* can execute only what the *spec*, in a genuine package implementation, would expose and can read only the anodyne pass-through-code to the partners in the *stopwatch_body* schema. This approach does, however, leak the information that the *stopwatch_body* schema exists. This is a theoretical drawback: it violates the letter of the principle of least privilege. But, because grantees like *client* are not given usage on the *stopwatch_body* schema, they can discover nothing about it beyond the fact that it houses subprograms with identical names and parameter lists to their partners in the *stopwatch* schema.

Discussion

The package notion in PL/SQL provides the best scheme with which to compare what PL/pgSQL supports for the reasons given earlier:

- The syntax and semantics are generally similar.
- The high-level purpose and the execution model are similar (tight, language-level integration with SQL; executes in the same process in which the SQL that it invokes executes).

Notwithstanding this, it's important to realize that the main arguments for enhancing PL/pgSQL to support a package notion have nothing to do with easing the migration of extant applications that use Oracle Database's PL/SQL to use PostgreSQL and PL/pgSQL instead, even though such an enhancement would doubtless help that endeavor.

Rather, the main arguments are simply to bring the generic benefits to PL/pgSQL programmers that programmers who use any language that supports a *module* construct enjoy.

I developed my case for bringing a package notion to PL/pgSQL in two parts:

- First, I showed working code (presented as a single self-contained, re-runnable script) for Oracle Database that anybody who has access to any version that was released during the past couple of decades can simply run. And then I showed my sketch of how this might be transliterated into PL/pgSQL. The code size is very similar. Of course, I've no idea how feasible it would be to support something along these line or how much programming effort it would take.
- Then I showed a runnable implementation that uses, as it must, free-standing PL/pgSQL subprograms distributed among two schemas with a common owner. This follows the approach that the PostgreSQL documentation recommends. It's also presented as a single self-contained, re-runnable script.

I tested the Oracle Database script using Version 18.4. And I tested the runnable PostgreSQL script using PG Version 14.1—and, for good measure using YugabyteDB Version 2.11 (which, in turn uses the SQL processing code of PG Version 11.2). The results are the same in all three environments.

The runnable PL/pgSQL approach suffers from a number of disadvantages with respect to a package approach. The following bullets are numbered to allow ease of reference. The order is insignificant.

(1) A package approach uses a documented language feature. Programmers can learn the relevant notions and syntax, once and for all, and then program in a tightly constrained, and therefore, uniform, way. This means that code authors don't need to invent their own conventions for package emulation and, especially, don't have to write external documentation to describe these conventions. Moreover, code that follows a convention is bound to differ between different development shops—and, in all likelihood, even within a single development shop. This makes maintenance hard.

(2) In the package approach, the helpers *fmt()* and *duration_as_text()* are hidden in the body by a whitelist notion that the language semantics brings: you don't want them to be usable except in the body; and so you don't ask for this. In contrast, because you do want the *start()* procedure and the *reading()* function to be usable by subprograms outside of the package, you ask for this explicitly by declaring them in the spec. In the runnable PL/pgSQL approach, you have to implement the whitelist by a convention-specified use of two schemas and appropriate privileges.

(3) The inner function support provides another language semantics mechanism for the programmer to express (especially to other readers) the intention that *fmt()* may be used only in the implementation of *duration_as_text()*. This intention could be expressed in the runnable PL/pgSQL approach by creating a dedicated user, for each subprogram that in the package approach would use inner subprograms, to own these helpers. The naming, and the grant statements, would express the programmer's intention. But, especially because the inner subprogram notion is defined recursively, this could rapidly get out of hand and subvert the intention of self-describing clarity of purpose.

(4) The package approach needs just two objects that, visibly and by definition, jointly act as a single construct. The runnable PL/pgSQL approach in the use case presented here needs two schemas, six explicit schema objects, and one implicit nonschema object—the user-defined run-time parameter. (You can discover that this parameter is part of the implementation only by reading the code.) However, real-world packages in PL/SQL often have on the order of ten subprograms exposed by the *spec*, maybe twice that number of *body*-private helper subprograms, and on the order of ten package global variables. So here, the *spec-body* pair would expand to require thirty-odd *create* statements together with ten-odd emergent run-time parameters. Reliance on a proliferation of very many schema objects and nonschema objects, with no self-describing scheme that they belong together except and externally documented practice convention, is a huge disadvantage.

(5) The general rule of good practice is that the *spec* should expose only variables marked *constant*. When it's intended that users of the package may directly change the value of a *body*-private global variable, this is done with a setter procedure, exposed by the *spec*, that typically ensures that the value that's set conforms to rules (as could be achieved, less directly, by defining a domain type for the variable). And when it's intended that users should access the present value of such a *body*-private global variable, this is done with a *spec*-level observer function. This setter/observer paradigm is the generic good practice for any language that supports a *module* notion for encapsulation.

The runnable PL/pgSQL approach that I used simply cannot emulate *body*-private global variables. The session's client can always set, and read, a user-defined run-time parameter if only it knows its name. (I do appreciate that *show all*, documented, implicitly, as "show the value of all run-time parameters", in fact shows surprisingly only the system-defined run-time parameters—even though *show foo.bar* does show the value of this user-defined run-time parameter.) This brings a security risk. Some exploits are conducted, in a secured production system, by disgruntled employees who were able to read the code in the development shop. Knowing the code doesn't help you violate the integrity of a genuine *body*-global variable. But that knowledge would let you change the value of a user-defined run-time parameter.

(6) A run-time parameter's value can only be *text*. This weakly-typed notion implies careful programming, in general, to typecast a to-be-set value to *text* and to typecast a to-be-read value to its intended data type.

(7) A temporary table, with an appropriate accompanying regime of privileges, could emulate strong typing and *body*-privacy. But it would require using *security definer* subprograms. However, the requirement might be to emulate a *security invoker* package. (As it turns out—see below—other considerations force the use of *security definer* subprograms.)

Moreover (at least as far as I have been able to discover), this would require that the client makes an explicit initialization call at the start of a session.

(8) The package notion brings a dedicated "hook" for initialization code: the executable section at the end of the *body*'s source text. This code runs just once in a session when an element in the package is first referenced. The runnable PL/pgSQL approach that I used cannot (as far as I can see) provide an emulation for this.

(9) The requirement that clients who can use the elements that a package *spec* exposes must not be able to read the code that defines the *body* forces the use of *security definer* subprograms. Else, each client must be given the explicit privilege to execute all of the subprograms that are intended to be *body*-private. And this inevitably allows reading the code. With a genuine package notion, there is no such spurious conflation of concerns: granting execute on an *authid current_user* package (this is how PL/SQL spells *security invoker*) does not confer the ability to read the text that defines its *body*.

Conclusion

I believe that I have presented an unassailable case for the value of extending PL/pgSQL to support a package notion. I cannot accept a counter argument that claims that PL/pgSQL programmers have no need for a *module* notion while the designers of very many other programming languages, where the notion is supported, have shown that they believe that the notion is not language specific and is generically useful. In particular, therefore, I cannot accept the notion that PL/pgSQL would benefit from packages only to ease migration from Oracle Database. I believe that the simple use case that I chose to illustrate this essay is an archetype for a vast range of use cases. The general experience of PL/SQL programmers, who can choose between free-standing subprograms and grouping these within packages, is that they almost always choose the package approach.

Of course, I would have to accept counter arguments like, say, the language design of PL/pgSQL, or of its execution model, mean that the die is irrevocably cast and it would be simply impossible to enhance it to support packages. I would also have to accept counter arguments based on the relatively huge implementation cost that such an enhancement would imply.

Bryn Llewellyn — bryn@yugabyte.com
Technical Product Manager
Yugabyte Inc., Sunnyvale, California, USA
January 2022
Bio (at "PostgreSQL Person of the Week")