# The pl/dotnet extension to PostgreSQL, v0.99(beta)

The pl/dotnet team
Brick Abode
Florianópolis Brazil
pldotnet@brickabode.com

## ABSTRACT

pl/dotnet extends the PostgreSQL database to support stored procedures, functions, triggers, and DO blocks for the .NET platform, including both C# and F#. In our benchmark it is the fastest Procedural Language in PostgreSQL, and it has the widest range of unit testing. It natively supports 38 out of 46 standard user types, the widest range of any external Procedural Language in PostgreSQL. It is released as free software under the PostgreSQL (BSD-style) license. Our goal is to be the best Procedural Language in PostgreSQL and the best implementation of .NET stored procedures in any database. We here present our work for consideration and feedback.

## KEYWORDS

PostgreSQL, .NET pl/dotnet, C#, F#, stored procedures, triggers

## 1 INTRODUCTION

PostgreSQL is an extensible, open-sourced object-relational database management system [4]. .NET (also "Dotnet") [3] is an open-sourced, cross-platform development framework including a JIT-compiler, a high-performance runtime, and support (including cross-calling) for multiple languages, including both C# (an object-oriented dialect of C) and F# (based on Ocaml, an object-oriented dialect of ML.)

The PostgreSQL project includes support for user-defined functions, stored procedures, and triggers. These can be implemented in a number of languages, including SQL, C, TCL, Perl, and Python in the standard distribution [4], and Java [1], Lua [5], and R [2] outside of the standard definition.

The pl/dotnet project extends PostgreSQL to support user functions, procedures, triggers, and `DO` blocks for the .NET platform, including both C# and F#. We support all Procedural Language features. We have achieved native representation for 38 of out of 46 PostgreSQL user types, plus their arrays, the widest range of any external Procedural Language in PostgreSQL. In our benchmarks we are the fastest Procedural Language in PostgreSQL, though only by a few percent.

## 2 MOTIVATION

Stored procedures had their moment in the sun in the late 1980s and early 1990s for three reasons of general awesomeness:

(1) they move the code to the data instead of the data to the code, which is faster and cheaper,
(2) they provide strong security, restricting data modification rights of clients, and
(3) they reduce the conceptual load on the database client, allowing pieces to be decoupled.

Stored procedures were then mostly discarded by our industry because of the practical problems in using them:

(1) writing stored procedures in a different language and software environment than the remainder of your codebase adds constraints and overhead,
(2) writing stored procedures in a different team than the remainder of your codebase adds overhead,
(3) stored procedure environments can't manage the software lifecycle (upgrades, etc) as well as normal software environments, and
(4) pl/sql, the dominant language for stored procedures is, as a technical matter, terrible and awful and very terrible.

This was a mistake. We have learned much in the past 30 years about how to address the problems, and the benefits are even more compelling today. Stored procedures can be great, and pl/dotnet will prove it.

.NET is a great platform for building servers. The .NET runtime is the best garbage-collected, multi-threaded runtime in the world, with good scalability and the ability to run on all modern operating systems. ASP.NET[1] is a powerful framework for building web apps.

.NET includes F#, a strongly-typed functional programming of the ML family. Functional programming goes naturally with relational database, which are in many ways inherently functional, and we have full support in pl/dotnet for F# as a first-class citizen.

PostgreSQL is the world's leading free software database, with a beautiful MVCC architecture, total SQL support, universal support across all major programming languages, and a strong community.

Our goal is to make pl/dotnet the best stored procedure languages for PostgreSQL and the best .NET stored procedure language in any database. Stored procedures will once again be a great option application builders.

## 3 PROJECT STATUS

As of this beta release, pl/dotnet supports all major procedural language features:

- Languages: C# and F# both fully supported
- Datatypes: 38 out of 46 PostgreSQL user (non-system) data types, plus their arrays, and all types are nullable
- Code can be entered directly via `CREATE FUNCTION` or, alternatively, loaded from pre-compiled assemblies
- Performance in our benchmarks surpasses all other PL implementations
- Testing: we have 1126 unit tests, covering all features in both C# and F#
- We imported all NPGSQL unit tests to measure our compatibility; 37.1%of them are working, 368/991
- Security: Code for each function is isolated in a .NET Assembly Load Context, providing nice security protection
- Full trigger support, including modifying data where allowed by SQL

[1] https://dotnet.microsoft.com/en-us/apps/aspnet

- Full support for output parameters (`OUT` and `INOUT`), nicely mapped to both C# and F#
- Full support for Set-Returning Functions
- Support for functions returning `RECORD`s and tables
- SPI support via the NPGSQL API, allowing much client code to be ported unmodified into server code

## 3.1 Our special relationship with NPGSQL

To make stored procedures great again, code needs to be maximally portable between the database client and the database server.[2] Any and all differences between the programming environment in the database client and the database server must be torn down. Ideally, the same code should be able to execute in either context with zero modification.

For this reason, API compatibility between the client and the server is paramount, and this has been and remains a major source of difficulty for developers in PostgreSQL. The problem is two-fold:

- For many languages, there is no single, canonical API for accessing PostgreSQL. This is the case for Python, where the major PostgreSQL APIs are the Django ORM, SQLAlchemy, and Psycopg. pl/python is compatible with none of them, instead exposing its own API for database access via SPI and data type mapping.
- Even when there is a canonical API, the procedural language environments do not use it. For example, PostgreSQL has a JDBC client library, but pl/java implements an entirely differeny JDBC API.

Dotnet has the age-old ADO.NET API, which so loosely defined that it is more a set of conventions than an API proper. Fortunately, however, there is a single, canonical client library for accessing PostgreSQL from dotnet: Npgsql.[3]

The presence of this universal API for PostgreSQL access in dotnet was a strategic gift to our project, and we were determined not to waste it. We not only adopted all of NPGSQL's mappings between dotnet types and PostgreSQL types; we also adopted NPGSQL as our abstraction layer over database access via the Server Programming Interface (SPI).[4]

When I say we "adopted the API", we did not merely reimplement it, as pl/java did to the JDBC library; we incorporated NPGSQL wholesale into our project. At the lowest levels of the library, we replaced its normal socket-based communication with the database server with direct calls to the SPI API. These modifications are very small and targeted, and they are entirely invisible to NPGSQL users.

You can see the advantage this approach gives us in Table 1, which compares our code size to that of other Procedural Language implementations in PostgreSQL[5][6]. This approach was more difficult for us in several ways, but it gave us nearly perfect API compatibiltiy, the widest range of native type support, and good performance while having such a small codebase; we are proud of this.

**Table 1: Lines of code for various PostgreSQL PL implementations**

| PL implementation | Lines of code |
|---|---|
| pl/java | 54984 |
| pl/v8 | 29526 |
| pl/pgsql | 13614 |
| pl/lua | 13008 |
| pl/python | 4535 |
| pl/r | 4413 |
| pl/dotnet | 4040 |
| pl/perl | 2741 |

I say "nearly perfect API compatibiltiy" because there are minor differences, forming a long tail of many small incompatibilities. The largest category is exception mapping, where the many different kinds of exceptions which NPGSQL throws require much work to map precisely to our SPI usage. Of course, throwing different exceptions is an API difference which needs to be addressed, but it is not a major difference, and this approach already gives our users a very high degree of compatibility.

We are confident of our ability to reach perfect compatibility in time, because we imported NPGSQL's entire regression test suite into pl/dotnet as stored procedures. Once we can pass all of these tests, then our compatibility will be at the same level which NPGSQL itself provides between version upgrades.

The data from these regression tests has been interesting. 37.1% of them are working, 368/991. The pattern we have observed, generally speaking, is that linear progress in the test suite corresponds to exponential progress in feature support: 80% feature support got us 20% test passage, 95% of feature support got us 40% test passage, etc. (These figures are highly informal.) The test failures are quite a thicket; we will fix a problem, and some of its failing tests are resolved, while others of its failing tests continue failing, but now with a new cause.

The current test passage rate indicates a high degree of compatibility with the calling of the API, while work remains in the smaller details of the API mapping. We have a long road ahead of us for the remaining unit tests, but we welcome it. This path will get us to not only full compatibility, but strong confidence as an engineering matter that the compatibility is precisely as strong as NPGSQL's own inter-version compatibility.

[2] A note on terminology: when we say "database client", we typically mean the trusted, centralized software component with privileged access to the database, which is more often called a "server", most often a web server. If we were to call it a "server" in this paper, then it would get confused with PostgreSQL itself, so we stick to the convention of referring to it as the database client, which it is. Such are the terminology problems of three-tier architectures.

[3] https://www.npgsql.org/doc/types/basic.html

[4] "The Server Programming Interface (SPI) gives writers of user-defined C functions the ability to run SQL commands inside their functions or procedures. SPI is a set of interface functions to simplify access to the parser, planner, and executor. SPI also does some memory management." - https://www.postgresql.org/docs/current/spi.html

[5] We here count code in C/C++ as well as the native language, with the exception of plpgsql; we do not count their PL-specific SQL code, which is unusually difficult to differentiate. Lines of code were counted using version 1.94 of the `CLOC` tool.

[6] We also made 1916 lines of changes to our fork of the the NPGSQL package. Most of these changes are boilerplate stemming from renaming and overriding the datatype classes. They are strictly external to pl/dotnet, so we did not count them. If you choose to count that way, the total is 5956 lines, bumping us up one place, past pl/python.

For this reason, only having 37.1%of the tests passing does not concern us; we consider it a good start in a very promising direction.

## 3.2 Data type support

PostgreSQL has a rich type system, with 46 user types in the main distribution. Any mapping of these types into a programming language is going to face choices and numerous challenges about how to map them, especially for intricate types such as datetimes. Our use of NPGSQL gave us a simple answer which is maximally useful for our users.

Table 2, which LaTeXhas yeeted somewhere in this document, lists the PostgreSQL data types support by pl/dotnet, which use exactly the same type mapping as Npgsql. (The type names are sometimes different in F#.)

pl/dotnet supports all of the range data types in PostgreSQL. We skipped multirange support for the time being and intend to add it in the future.

*3.2.1 Arrays and Nulls.* All supported data types also support arrays of that type, be they single-dimensional or multi-dimensional. We currently do that via the Npgsql convention of mapping them to `Array<Type>`, but that type mapping is cumbersome and expensive compared to the somewhat different `Type[]` representation.

The source of this problem is an unfortunate design choice in PostgreSQL, which tracks types according to their `OID`. Each datatype has a corresponding array datatype, with its own OID, but this OID does not encode the dimensionality of the array; thus, all arrays in PostgreSQL may be of arbitrary dimension[7], regardless of which dimension they were declared with. You can see this reflected in the PostgreSQL manual:

> The syntax for CREATE TABLE allows the exact size of arrays to be specified, for example:
>
> ```
> CREATE TABLE tictactoe (
>     squares integer[3][3]
> );
> ```
>
> However, the current implementation ignores any supplied array size limits, i.e., the behavior is the same as for arrays of unspecified length.
> The current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring the array size or number of dimensions in CREATE TABLE is simply documentation; it does not affect run-time behavior.
>
> **Postgresql Manual, Ch. 8.15, Arrays**

[7] Up to the hard limit ("`MAXDIM`") of 6.

Because Npgsql cannot know the dimensionality of the arrays it will receive in a client context, it must use an object representation which can handle any dimensionality; this is cumbersome in development and slow in execution. However, since we are operating in a stored procedure context instead of a client context, we have another option available to us.

All types in PostgreSQL are nullable by default, so our default handling is to map parameters to the nullable type (`T?`) in .NET . This is similar to a `Option` type in ML and F#, but sadly not the same. To keep ourselves For PostgreSQL functions that are defined as `STRICT`, the function will not be invoked with null values, so in that case we map the type to the simpler `T` type. We hope to make this behavior configurable for the developer in the future.

## 3.3 Parameter Modes

pl/dotnet supports all three SQL parameter modes: `IN`, `INOUT`, and `OUT`.

For C#, we handle `IN` arguments normally. `INOUT` parameters are typed as `ref` arguments, which maps `INOUT` behavior cleanly to C#. `OUT` parameters are simply mapped to C#'s `out`.

Such handling of parameters, though idiomatic in C#, are very out of place in functional programming languages, so in F# we pass `IN` and `INOUT` values as input arguments, and all `INOUT` and `OUT` values are separately returned in a tuple. Ironically, this is how the variables are actually processed in PostgreSQL's internal handling, while our C# mapping matches the SQL syntax.

## 3.4 Function, Procedure, DO, Trigger

PostgreSQL has four modes in which code can be invoked from a Procedural Language:

(1) As a user function ("`CREATE FUNCTION`"), taking parameters, returning a value, but not allowing database modifications
(2) As a procedure ("`CREATE PROCEDURE`"), taking arguments, not returning a value, but allowing database modifications
(3) As a "`DO`" block, creating a transient anonymous function in a Procedural Language.
(4) As a trigger ("`CREATE TRIGGER`"), which is a function called when certain database events happen.

pl/dotnet supports all four modes.
Our trigger support includes all trigger operations:

(1) Trigger arguments
(2) Full trigger information: event, level, table name, operation, etc.
(3) Full copies of old and new rows
(4) Ability to modify the row, when allowed under SQL

## 3.5 Set-Returning Functions (SRFs) and Records

Set-Returning Functions (SRFs) and Records are both features of interest.

SRFs were nicely mapped to the native conventions in C# and F#. In C#, they are mapped to enumerators:

```
CREATE OR REPLACE FUNCTION
make_pi()
RETURNS SETOF float8 AS
$$
    // In C#, this maps to:
    // public static IEnumerable<double?> make_pi()
    double sum = 0.0;
    for(int i=0;;i++){yield return
    ↪  4*(sum+=((i%2)==0?1.0:-1.0)/(2*i+1));}
$$
LANGUAGE plcsharp;
```

In F#, they are mapped to sequences, which share the same IEnumerable dotnet interface as C# enumerators:

```
CREATE OR REPLACE FUNCTION
make_pi_fsharp()
RETURNS SETOF float8 AS
$$
    // In F#, this maps to:
    // static member make_pi_fsharp() : seq<Nullable<double>> =
    seq {
        let mutable sum : float = 0.0
        for i = 0 to System.Int32.MaxValue do
            yield double(4.0 * sum)
            sum <- sum + ((if i % 2 = 0 then 1.0 else -1.0)/
            ↪  float(2.0 * float(i) + 1.0))
    }
$$
LANGUAGE plfsharp;
```

Records were another interesting feature. The nature of a record in SQL is that it can hold any data type, so we represent it as an array of type Object, the universal type in dotnet.

```
CREATE OR REPLACE FUNCTION
dynamic_record_generator_srf(lim INT8)
RETURNS SETOF record
AS $$
    // In C# this maps to:
    // public static IEnumerable<Object? []?>
    ↪  dynamic_record_generator_srf(long? lim)
    if (!(lim > 0)){ yield break; }
    for(long i=0;i<lim;i++){ yield return new object?[] { (long)i,
    ↪  $"Number is {i}" }; }
$$
LANGUAGE plcsharp;
```

```
CREATE OR REPLACE FUNCTION
dynamic_record_generator_srf_fsharp(lim INT8)
RETURNS SETOF record
AS
$$
    // In F# this maps to:
    // static member dynamic_record_generator_srf_fsharp (lim:
    ↪  Nullable<int64>) : seq<obj[]> =
    match lim.HasValue with
    | false ->
        seq { for i in 0 .. System.Int32.MaxValue do yield [| box
        ↪  i; $"Number is {i}" |] }
    | true ->
        if not (lim.Value > 0) then
            seq { () }
        else
            seq { for i in 0L .. lim.Value - 1L do yield [| box i;
            ↪  $"Number is {i}" |] }
$$
LANGUAGE plfsharp;
```

Architecturally, records were very interesting for us, because they are the one case where the type is not known at compile-time. It is a major advantage of our architecture that we resolve types at compile-time instead of run-time, allowing us to generate code with the minimal execution path instead of having to dynamically dispatch the types on each call. However, with records, this was impossible, so we also implemented the same kind of dynamic type lookup which the traditional PLs do, but we only use it in the dynamic case. We were even able to detect type mismatches between what dotnet returns and what the database is expecting and handle them.

## 3.6 Using Both Code and Assemblies

Traditional PostgreSQL Procedural Languages, such as pl/pgsql and pl/python, support creating user functions by passing the code as part of the declaration of the function.

Alternatively, PostgreSQL's pl/java, along with the .NET (CLR) implementations in Microsoft's SQLServer and IBM's DB2, support loading a function from a pre-compiled assembly ("`.dll`" or "`.jar`") file. PostgreSQL also allows compiled (C, rust-lang, etc.) functions to be loaded from shared library files.

pl/dotnet supports both modes. First, here are examples of directly entering the code in both C# and F#:

```
CREATE OR REPLACE FUNCTION IntegerTestCS(a INT4, b SMALLINT)
RETURNS INT4 AS $$
  return a+b;
$$ LANGUAGE plcsharp STRICT;

CREATE OR REPLACE FUNCTION IntegerTestFS(a INT4, b SMALLINT)
RETURNS INT4 AS $$
  a+b;
$$ LANGUAGE plfsharp STRICT;
```

Second, here is an example of loading the function from a DLL, which shoud work for any .NET language and is tested for C# and F#:

```
CREATE OR REPLACE FUNCTION IntegerTest(a INT4)
RETURNS INT4 AS 'Sample.dll:Namespace.Class!IntegerTest'
LANGUAGE plcsharp STRICT;
```

PostgreSQL does have support for declaring the library from which to load a function, but only for natively-compiled (that is, C) functions. For other Procedural Languages, the PostgreSQL parser currently does not pass the file location to the language handler, as this was not an anticipated use case. For newer language handlers like pl/dotnet and pl/java, which also support loading code from external libraries/archives/assemblies, this would be a nice feature for PostgreSQL to provide, and we might add it.

## 3.7 Platform support

*3.7.1 Operating systems and CPU.* pl/dotnet is primarily developed on Linux and has been fully tested on both x86 and ARM CPUs, suggesting the absence of any obvious endianness bugs. We also have built and tested pl/dotnet on MacOS (OSX) on ARM, though it is of secondary priority to us.

Our build environment is container-based, allowing us a stable and repeatable build environment. We build and distribute Debian packages and Docker images.

There should be no problems getting pl/dotnet running on Windows; there are few system dependencies in the code, and those are very standard.

We welcome code contributions to add support on other Linux distributions and on other operating systems for pl/dotnet.

*3.7.2 Postgresql versions.* We support PostgreSQL versions 10, 11, 12, 13, 14, and 15. Currently, all features are supported for all PostgreSQL versions, and the only anticipated exception to that support is multirange, which was added in PostgreSQL v14, and which we intend to add support for soon.

*3.7.3 .NET versions.* We currently develop against .NET version 6. Support for other versions of .NET should also not be difficult; this will be an area of work after our 1.0 release.

## 3.8 Security

pl/dotnet has reasonable security. We use separate Assembly Load Contexts[8] for each stored procedure, which provides some level of isolation between them: they exist in separate namespaces and generally do not have access to each other's code or data, or that of the underlying system.

Thus, there is no straightforward way for stored procedures to interfere with PostgreSQL, but it might be possible to do so with not-straightforward paths. We could marginally improve the security of the system by reducing the set of libraries available to stored procedures, but even this would not be a guarantee.

Microsoft previously attempted to provide these guarantees in dotnet with AppDomain[9] but eventually gave up and (rightly) deprecated it. The number of potential avenues of attack are simply too great to be able to secure with certainty unless the platform is designed from the ground up to provide such security, and almost no modern language runtimes were engineered in that way.

This problem is faced by every stored procedure language, and unless the language provides air-tight guarantees as to its security, then we think that skepticism regarding such assurances is warranted. Few languages in existence provide such guarantees, and we do not think that any of the current stored procedure languages qualify. Only pl/tcl even makes such a claim, and while we respect their design, we would still probably not fully trust it for security-critical usage.

The fundamental problem is that stored procedures execute inside of the PostgreSQL server's memory space, and the operating system and CPU provide no memory protection between the stored procedures and the database. This is the fundamental tradeoff to be made in order to get the increased performance which stored procedures provide.

Even with these limitations, we think there is a security argument to be made for this approach over the traditional architecture. In the normal use case, database clients cannot easily interfere with PostgreSQL's internal operation, but they have access, usually unrestricted, to modify or delete the data as well as the schema.

We here should think about why PostgreSQL's integrity is important. It is unusual that an attacker wishes to use one application to subvert the server in order to access to another database, because most databases are not shared in the modern environment. Thus, the only function which is served by PostgreSQL's integrity is to enforce the security restrictions on the application's database.

In an environment where the client has unfettered read/write access to the database, and that is the common case, the integrity of PostgreSQL is not important, because an attacker who has subverted the client already has full access.

It is generally easier to subvert the security of a client process, for example in a public-facing web server, than it is to subvert the security of the database or the stored procedures which it holds. By trusting a limited set of code inside of the database, it is possible to dramatically limit the trust one needs to extend to the database client, which is generally larger, more exposed, and more difficult to secure. Because the client no longer needs unrestricted write access to the database, this change will usually yield an improvement in overall security, often a dramatic one.

For this reason, we believe our design to offer superior security for many classes of users.

Stored procedure authors must be the final arbiters of their security tradeoffs. They should, as always, take care that their code not introduce means to be subverted by malicious input. This is easier in a memory-safe environment like .NET .

Security-critical applications do exist, and we are interested in serving them; it is an interesting area for future work.

## 3.9 Code Quality

In order to improve our code quality, we have built unit tests for all supported datatypes, their arrays, and null handling in both C# and F#, as well as for all major functionality, such as triggers, Set-Returning Functions, table functions, etc.

We use three static analysis tools:

(1) Cpplint[10] is a static code checker for C and C++.
(2) StyleCop[11] is a static code analysis tool. Formerly a standalone tool, it was refactored to become a series of Roslyn plugins.
(3) SonarLint[12] is a code quality and security static analysis tool with almost 5000 rules.

C and C# code in pl/dotnet is clean under the cpplint, StyleCop, and SonarLint checks.

Source code in pl/dotnet, both C and C#, is documented using Doxygen, and the generated documents are reasonably complete in explaining the system.

Finally, we imported all of NPGSQL's unit tests to run under pl/dotnet, and we have several hundred of them working. The NPGSQL unit tests failures are caused first by minor differences in exception handling, and also by our incomplete support for NPGSQL's feature set. We continue to make progress in improving our NPGSQL compatibility, but we believe that the current feature set is very useful, and our using NPGSQL's own tests leaves us confident that our supported features are supported well.

[8] https://learn.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=net-7.0
[9] https://learn.microsoft.com/en-us/dotnet/core/porting/net-framework-tech-unavailable
[10] https://github.com/cpplint/cpplint
[11] https://github.com/DotNetAnalyzers/StyleCopAnalyzers
[12] https://rules.sonarsource.com/csharp

### 3.10 Future Features

We do not currently support dynamic database types, such as enums, composites, domain types, and other user-defined types. Npgsql does support them, and we intend to add them in the future, but their handling is somewhat delicate, so we chose to ship the first version without them. We also do not support the Numeric type.

We plan to give developers more control over code isolation by placing each DLL in its own AssemblyLoadContext. Thus, code from the same DLL can share state, while functions are otherwise isolated from each other. This will let users control data sharing between their functions by controlling which DLLs hold their functions.

Our automated tests cover not only C# and F#, but also numerous other stored procedure languages. We hope to share these for use by other PL teams and to facilitate cooperation across the PostgreSQL PL community. We would like for our project to be helpful to other PL's in improving their implementations, and we thank them for the help which their examples provided us in developing this project.

## 4 .NET IMPLEMENTATION

### 4.1 Architecture

Any stored procedure language in PostgreSQL is going to have some mix of C code and language-specific code, and there are choices to be made in where and how you build functionality. We liked the increased safety of working in C#, and we valued the richer set of programming tools, so our decision was to keep the C code to a minimum and use C# wherever possible handle the transfer of data into and out of user functions.

This stands in contrast to some other PL implementations, and pl/python is an interesting example. pl/python is implemented entirely in C, and we have learned much from it. The Python C API is very good, but it is still C. Having been built in C, pl/python must pay careful attention to manual memory management, and building objects is tedious. Despite our having a much wider range of native data type support than pl/python, we only have 25% as many lines of C code, 1123 versus 4533. Since C# is still compiled via the .NET JIT, and the CLR must be used anyway, we do not sacrifice much performance this way. Further, with our code generation strategy, we actually have a superior execution path to C engines such as pl/python's; whereas they are structured more like an interpreter, with run time determination of the execution path, we are able to lift such decision making to compile time and build minimal execution paths. Our optimized execution path is then compiled, yielding roughly equivalent and sometimes superior performance to natively-compiled code. For these reason, our performance is superior, despite building most of our logic in a high-level language. Keeping most handling in C# was thus a good design decision for us.

Every piece of data in PostgreSQL is expressed in a datum, and each datum has an Object IDentifier ("`OID`") for its type. (This OID is not in the datum at run time; it is in the type definition for the function at compile time.) For each data type, pl/dotnet has a pair of small C function whose purpose is to get the values out of the datum to .NET (called `InputValue()`) and then back from .NET into the datum (called `OutputValue()`.) All of the remaining handling is then done in C#. This also has the nice effect of keeping both C code and unsafe C# code to a minimum; they are strictly limited to getting data into and back out of .NET .

When a function is created, pl/dotnet creates two assemblies: the UserHandler, and the UserFunction. The UserFunction is a minimal wrapper around the user's code, or in cases where the user has loaded the function from a pre-compiled assembly, then it is a wrapper around that assembly. The type mapping from PostgreSQL to C# or F# is automated; the user need not bother himself with it. The UserHandler is responsible for marshalling values out of the database, calling the UserFunction, and then returning the result values back into the database.

A nice feature of this architecture is that, because the UserHandler is so cleanly separated from the UserFunction, support for pre-compiled DLLs was straightforward. Further, it makes F# support easy, because we can reuse the C# implementation of the UserHandler and need only compile the UserFunction in F#, which is then easily callable from C#.[13]

When a function is called from PostgreSQL, C builds an array of datums[14] with the function arguments and passes them to C#. C# knows the type of each argument at compile time, and it calls the precise handlers for each type to convert it from a PostgreSQL datum to a .NET value, without any run-time overhead. The C# handler handles NULLs and arrays with code that is nicely generic.

After those values have been passed to the user function, the return value from the user function then follows the reverse path through the type handlers to create a PostgreSQL result datum, which is then returned to the database.

### 4.2 Compilation and caching

Both C# and F# make use of a template source file which is then customized to create the source code for each user function. Users can optionally inspect the generated code.

pl/dotnet uses the the .NET Compiler Platform SDK, aka Roslyn[15][16], which is a set of compiler tools to compile C# to an in-memory assembly. Roslyn exposes the entire compiler pipeline to the application, providing us various features when compiling user functions, including:

- extensive code checking
- informative error messages, including correct line numbers
- ability to rewrite the code in the AST if needed
- code inspection, including metadata
- fine-grined control over library availability

We formerly used F# Compiler Services ("FCS")[17] in a similar way to dynamically compile F# user functions into an assembly, but compatibility problems forced us to switch to external compilation, which is slower at `CREATE FUNCTION` time.

These facilities give us many tools for improving the developer experience and have been essential to features such as detailed error messages and proper line numbering.

[13] The details of this handling are still evolving for F#, because of the differences between F# Compiler Services and Roslyn.
[14] We here refer to the plural of a PostgreSQL type "Datum" as "Datums" in order to technically differentiate them from the more generic term "Data".
[15] https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/
[16] https://github.com/dotnet/roslyn
[17] https://fsharp.github.io/fsharp-compiler-docs/fcs/

The generated code is then compiled into in-memory assemblies. If a pre-compiled assembly/DLL has been used, then that is loaded alongside the generated UserHandler. For F#, we support doing this with both dynamic assemblies and normal assemblies. These assemblies are loaded into an Assembly Load Context, which provides limited isolation of one function from another and also from the core system.

The resulting assembly context is then cached for reuse on subsequent calls. If the function has been dropped from the cache, either by a cache replacement or by a database restart, then we will transparently re-compile and the function the next time that it is called, again caching the result.

## 4.3 Strings

pl/dotnet currently assumes that all PostgreSQL strings are encoded in utf8, which is a superset of ASCII. PostgreSQL supports other character encodings, and we could extend our support for them, but utf8 is the de facto standard today, so alternative encodings are mostly of historical interest, making this functionality not urgent.

.NET spans let you create an object from only a pointer and a length, leaving the contents where they lie rather than requiring that they be copied.[18] Strings in pl/dotnet are handled via a `ReadOnlySpan`, which is an inexpensive and nice interface.

## 4.4 Applying PL/DOTNET with C#

We present a sample function to show the different parts of the system working together. We use Int16 and Int32 as simple example types, letting us concentrate on the control flow rather than the complexity of the type conversion. Some of the minor details have been simplified for presentation.

First is the SQL definition of the function; this is how the user will create the function in PostgreSQL.

```
CREATE OR REPLACE FUNCTION IntegerTest(a INTEGER, b SMALLINT)
RETURNS INTEGER AS $$
    return a+b; // this is the user code
$$ LANGUAGE plcsharp STRICT;
```

**Program code 1: How to define a function**

Figure 2 is a UML sequence diagram explaining the relative calls between PostgreSQL, the C portion of pl/dotnet, the C# portion of pl/dotnet, and the user-supplied function.

Program Code 2 is the generated C# code. Datum, which is handled as a `void*` in C, is handled as an `IntPtr` in C#.

[18] https://learn.microsoft.com/en-us/dotnet/standard/memory-and-spans/

```
namespace PlDotNET.UserSpace {
 public static class UserFunction {
   public static int? integertest(int a, short b) {
#line 1
     return a + b; // this is the user code
   }
 }
 public static class UserHandler {
  public static IntHandler IntHandlerObj = new IntHandler();
  public static ShortHandler ShortHandlerObj = new ShortHandler();
  public static unsafe void CallUserFunction(List<IntPtr>
  ↪  arguments, IntPtr output, bool[] isnull) {
   var argument_0 = IntHandlerObj.InputValue(arguments[0]);
   var argument_1 = ShortHandlerObj.InputValue(arguments[1]);
   var result = PlDotNET.UserSpace.UserFunction.integertest(
   ↪  (int)argument_0, (short)argument_1);
   var resultDatum = IntHandlerObj.OutputNullableValue(result);
   OutputResult.SetDatumResult(resultDatum, result == null,
   ↪  output);
  }
 }
}
```

**Program code 2: Generated C# code**

Program Code 3 is the generic conversion code, which handles NULLs and wraps the type-specific conversion code.

```
public abstract class StructTypeHandler<T> : BaseTypeHandler<T>
↪  where T : struct {
 public T? InputNullableValue(IntPtr datum, bool isnull) {
  return isnull ? null : this.InputValue(datum);
 }
 public IntPtr OutputNullableValue(T? value) {
  return value == null ? IntHandler.pldotnet_CreateDatumInt32(0) :
  ↪  this.OutputValue((T)value);
 }
}
```

**Program code 3: Generic conversion code**

Program Code 4 is the relevant integer-specific conversion code.

```
public class IntHandler : StructTypeHandler<int> {
 [DllImport("@PKG_LIBDIR/pldotnet.so")]
 public static extern int pldotnet_GetInt32(IntPtr datum);
 [DllImport("@PKG_LIBDIR/pldotnet.so")]
 public static extern IntPtr pldotnet_CreateDatumInt32(int value);

 public override int InputValue(IntPtr datum) {
  return pldotnet_GetInt32(datum);
 }
 public override IntPtr OutputValue(int value) {
  return pldotnet_CreateDatumInt32(value);
 }
}
```

**Program code 4: Main integer-specific conversion code**

Program Code 5 is the C code, which uses the relevant PostgreSQL macros (etc) for Datum conversion.

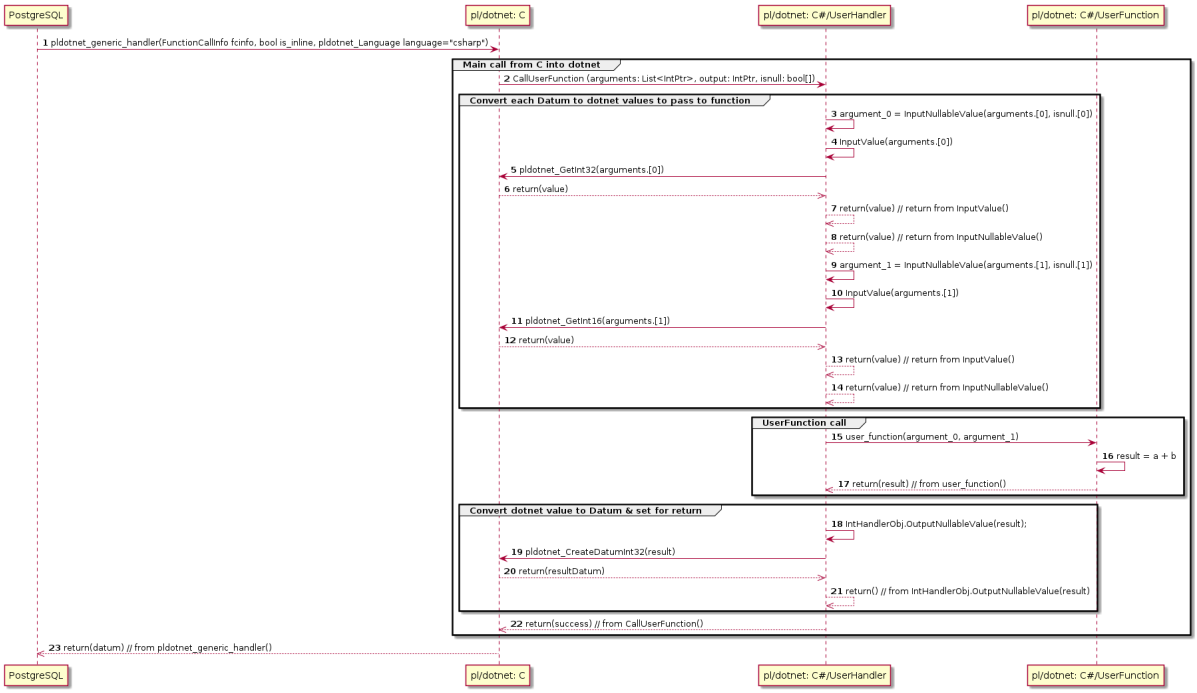



**Figure 2: Figure 2: pl/csharp Sequence Diagram**

```
int32_t pldotnet_GetInt32(void *datum) {
 return DatumGetInt32((Datum)datum);;
}
Datum pldotnet_CreateDatumInt32(int32_t value) {
  return Int32GetDatum(value);
}
```

**Program code 5: C code with PostgreSQL macros**

As you can see, the minimum of processing is done in C, and everything else is handled in C#. For this reason, each datatype has a pair of C handlers (input and output) in "`src/pldotnet_conversions.c`" and a corresponding pair of C# wrappers in "`dotnet_src/TypeHandlers/`".

## 5 RESULTS AND ANALYSES

In addition to our own tests for pl/dotnet, we have built automated testing for over a hundred features across a range of other PostgreSQL Procedural Languages:

We intend to assemble these tests into a unified suite which can be used by the entire PostgreSQL PL community. We hope that this can help us all improve performance, improve cooperation across the PL space, and help us work together to drive improvement in PostgreSQL to support the PL implementations.

### 5.1 Performance

We used these tests for benchmarking pl/dotnet and the other Procedural Languages. We present these results here.

It is important first to say that these benchmarks were not designed to fairly (or unfairly) evaluate the performance of other languages. Rather, they were intended to help us explore other languages' handling of PostgreSQL datatypes, understand pl/dotnet's performance, and find our own problems. They cover a wide range of stored procedure types and functionality, but they are not intended (and probablly cannot be) representative of the actual performance experienced by users, which will of course depend on the particular features which they use. We merely built the tests we needed for comparison, ran them, and benchmarked pl/dotnet's performance against them.

Further, these tests are designed measure the overhead of the PL engine itself. Of course, the performance of JIT-compiled platforms like Java and .NET will be significantly higher than interpreted languages like python and tcl; these benchmarks are not intended to measure such runtime differences.

We wrote the benchmarks at the beginning of the project, and the only modification which we made was to reduce the Fibonacci test, because it was making pl/dotnet look too good, destroying the visibility in our heatmap. (pl/fsharp dominated this test, which is unsurprising and also made us smile, but we shrank it anyway.)

To compute total performance, we equally weighted each test and compared pl/dotnet against the other PL in question.

Under these tests, pl/csharp is the fastest PL, and pl/fsharp is a close second. pl/pgsql is third. Performance among the top five languages is comparable; we do not massively outperform the other languages.

First, a summary:

Figure 3 shows the graphs of the relative performance of each language. Figure 4 is the more detailed heatmap showing relative performance for each test.

It is worth noting that, in actual usage, we expect pl/dotnet to be significantly faster than the interpreted languages such as pl/pgsql

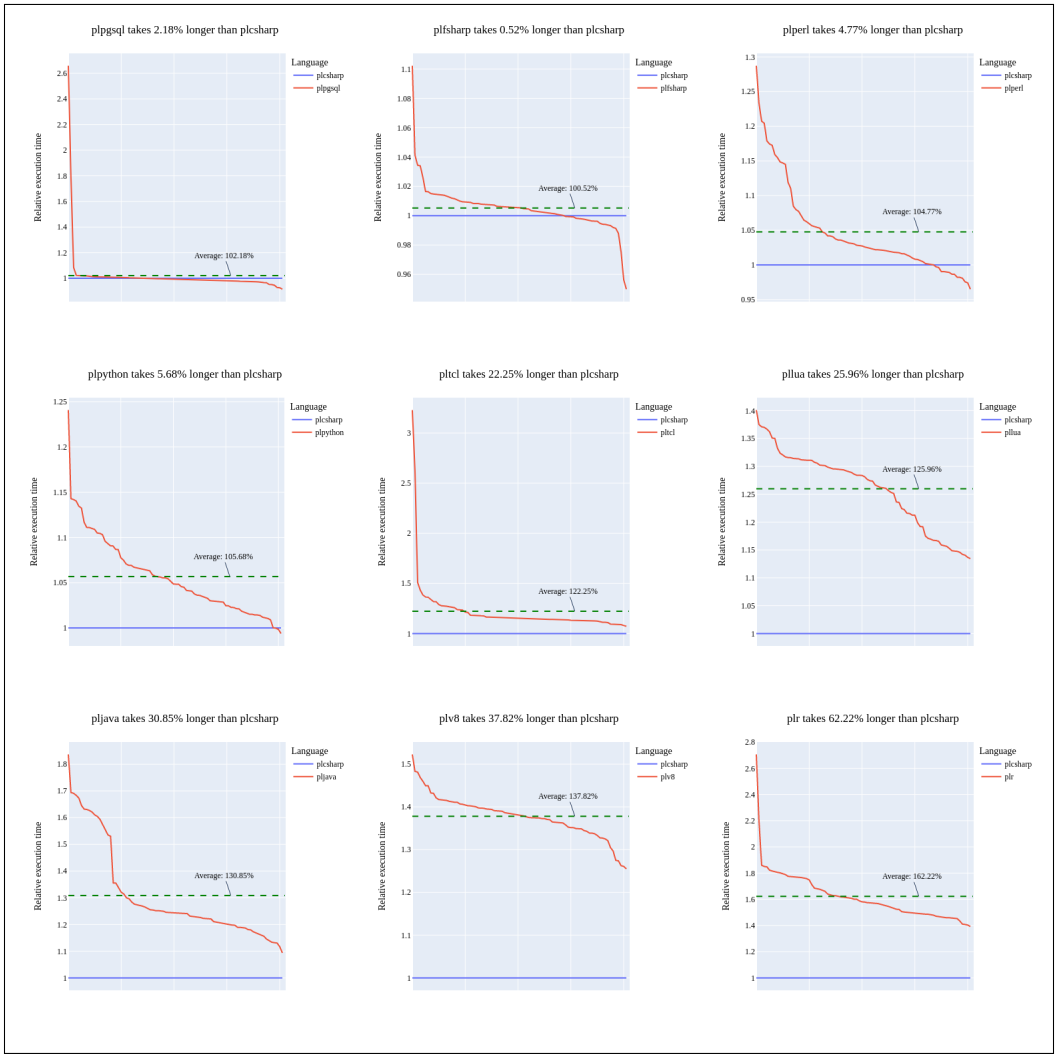



**Figure 3: Performance graphs for pl/csharp compared to other PostgreSQL Procedural Languages**

because of the superiority of the dotnet runtime. We might add performance tests to show this at a later date.

In the final stages of our project, we did some minor optimizations of our code base, specifically on arrays and string processing, and these benchmarks were helpful to us in focusing our efforts on optimizations that would be important.

## 5.2 Type support

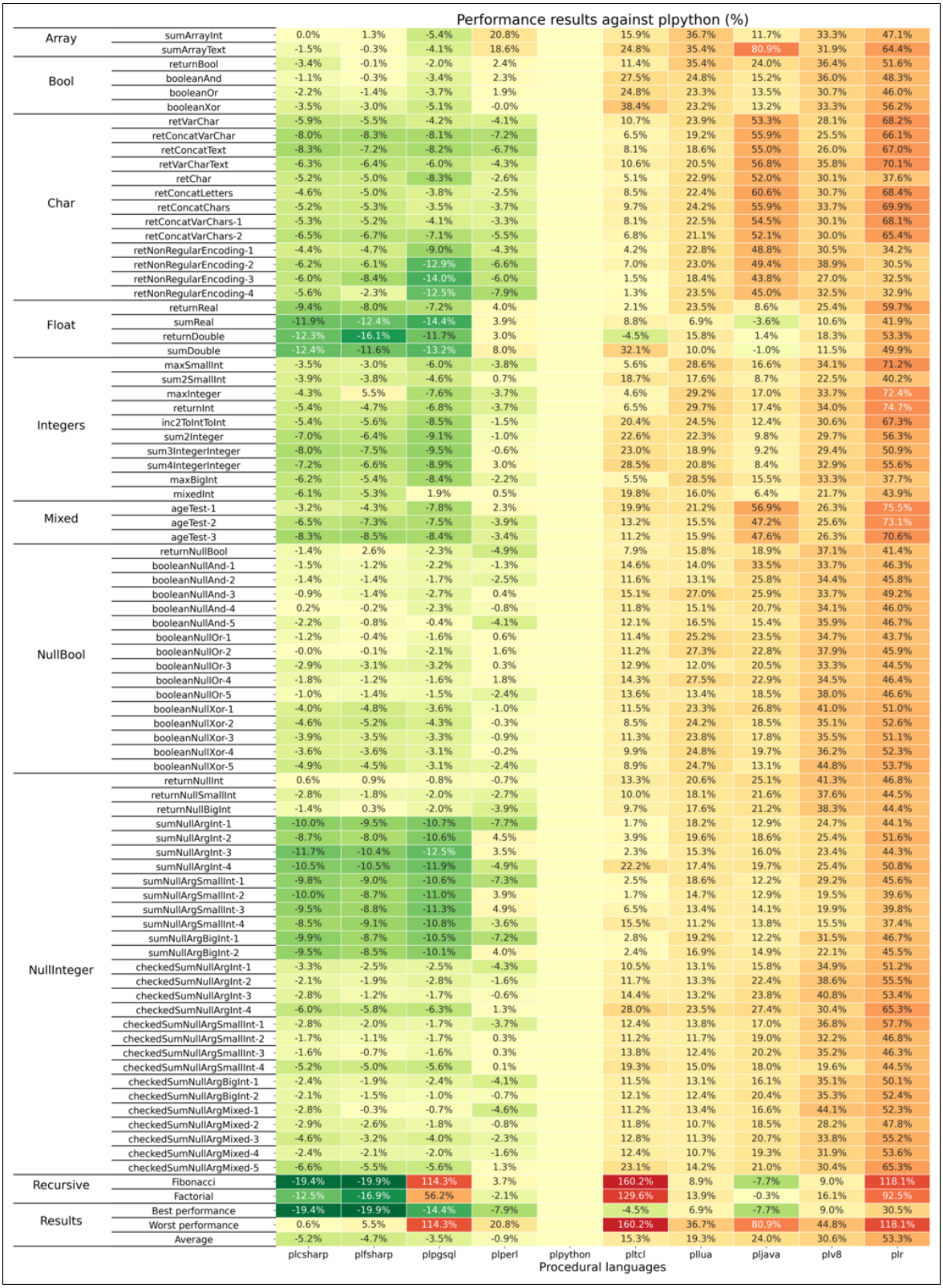
We consider 46 PostgreSQL types to be non-system types, intended for users. Of these, we support 38 for them, and we have a clear roadmap to support all 46 .

Numerous of the PL implementations for scripting languages, such as pl/python and pl/tcl, achieve type support by passing the PostgreSQL string representation to their functions for most or all data types. This nominally achieves full type support, but at the cost of pushing ambiguity and processing work onto the developer, as well as significant run-time overhead. We do not consider this to be "native" support.

pl/dotnet and pl/java map PostgreSQL values into their platform's corresonding native type, with a rich set of operators for each one. Here, we were greatly aided by the existence of Npgsql, which already has extensive mapping of the PostgreSQL type system to the .NET type system. Leveraging their work, we are able to have the greatest range of native type support of any external Procedural Language in PostgreSQL.

We still have four gaps in our type coverage: the Numeric type; composite and table types; enumerated types ("ENUM"); and multirange support for all five range types. (Multirange will be a single, generic implementation.)

We indend to add all of these, which would make us the first external Procedural Language with 100% native support of all PostgreSQL user types.




| | | plcsharp | plfsharp | plpgsql | plperl | plpython | pltcl | pllua | pljava | plv8 | plr |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Array | sumArrayInt | 0.0% | 1.3% | -5.4% | 20.8% | | 15.9% | 36.7% | 11.7% | 33.3% | 47.1% |
| | sumArrayText | -1.5% | -0.3% | -4.1% | 18.6% | | 24.8% | 35.4% | 80.9% | 31.9% | 64.4% |
| Bool | returnBool | -3.4% | -0.1% | -2.0% | 2.4% | | 11.4% | 35.4% | 24.0% | 36.4% | 51.6% |
| | booleanAnd | -1.1% | -0.3% | -3.4% | 2.3% | | 27.5% | 24.8% | 15.2% | 36.0% | 48.3% |
| | booleanOr | -2.2% | -1.4% | -3.7% | 1.9% | | 24.8% | 23.3% | 13.5% | 30.7% | 46.0% |
| | booleanXor | -3.5% | -3.0% | -5.1% | -0.0% | | 38.4% | 23.2% | 13.2% | 33.3% | 56.2% |
| Char | retVarChar | -5.9% | -5.5% | -4.2% | -4.1% | | 10.7% | 23.9% | 53.3% | 28.1% | 68.2% |
| | retConcatVarChar | -8.0% | -8.3% | -8.1% | -7.2% | | 6.5% | 19.2% | 55.9% | 25.5% | 66.1% |
| | retConcatText | -8.3% | -7.2% | -8.2% | -6.7% | | 8.1% | 18.6% | 55.0% | 26.0% | 67.0% |
| | retVarCharText | -6.3% | -6.4% | -6.0% | -4.3% | | 10.6% | 20.5% | 56.8% | 35.8% | 70.1% |
| | retChar | -5.2% | -5.0% | -8.3% | -2.6% | | 5.1% | 22.9% | 52.0% | 30.1% | 37.6% |
| | retConcatLetters | -4.6% | -5.0% | -3.8% | -2.5% | | 8.5% | 22.4% | 60.6% | 30.7% | 68.4% |
| | retConcatChars | -5.2% | -5.3% | -3.5% | -3.7% | | 9.7% | 24.2% | 55.9% | 33.7% | 69.9% |
| | retConcatVarChars-1 | -5.3% | -5.2% | -4.1% | -3.3% | | 8.1% | 22.5% | 54.5% | 30.1% | 68.1% |
| | retConcatVarChars-2 | -6.5% | -6.7% | -7.1% | -5.5% | | 6.8% | 21.1% | 52.1% | 30.0% | 65.4% |
| | retNonRegularEncoding-1 | -4.4% | -4.7% | -9.0% | -4.3% | | 4.2% | 22.8% | 48.8% | 30.5% | 34.2% |
| | retNonRegularEncoding-2 | -6.2% | -6.1% | -12.9% | -6.6% | | 7.0% | 23.0% | 49.4% | 38.9% | 30.5% |
| | retNonRegularEncoding-3 | -6.0% | -8.4% | -14.0% | -6.0% | | 1.5% | 18.4% | 43.8% | 27.0% | 32.5% |
| | retNonRegularEncoding-4 | -5.6% | -2.3% | -12.5% | -7.9% | | 1.3% | 23.5% | 45.0% | 32.5% | 32.9% |
| Float | returnReal | -9.4% | -8.0% | -7.2% | 4.0% | | 2.1% | 23.5% | 8.6% | 25.4% | 59.7% |
| | sumReal | -11.9% | -12.4% | -14.4% | 3.9% | | 8.8% | 6.9% | -3.6% | 10.6% | 41.9% |
| | returnDouble | -12.3% | -16.1% | -11.7% | 3.0% | | -4.5% | 15.8% | 1.4% | 18.3% | 53.3% |
| | sumDouble | -12.4% | -11.6% | -13.2% | 8.0% | | 32.1% | 10.0% | -1.0% | 11.5% | 49.9% |
| Integers | maxSmallInt | -3.5% | -3.0% | -6.0% | -3.8% | | 5.6% | 28.6% | 16.6% | 34.1% | 71.2% |
| | sum2SmallInt | -3.9% | -3.8% | -4.6% | 0.7% | | 18.7% | 17.6% | 8.7% | 22.5% | 40.2% |
| | maxInteger | -4.3% | 5.5% | -7.6% | -3.7% | | 4.6% | 29.2% | 17.0% | 33.7% | 72.4% |
| | returnInt | -5.4% | -4.7% | -6.8% | -3.7% | | 6.5% | 29.7% | 17.4% | 34.0% | 74.7% |
| | inc2ToIntToInt | -5.4% | -5.6% | -8.5% | -1.5% | | 20.4% | 24.5% | 12.4% | 30.6% | 67.3% |
| | sum2Integer | -7.0% | -6.4% | -9.1% | -1.0% | | 22.6% | 22.3% | 9.8% | 29.7% | 56.3% |
| | sum3IntegerInteger | -8.0% | -7.5% | -9.5% | -0.6% | | 23.0% | 18.9% | 9.2% | 29.4% | 50.9% |
| | sum4IntegerInteger | -7.2% | -6.6% | -8.9% | 3.0% | | 28.5% | 20.8% | 8.4% | 32.9% | 55.6% |
| | maxBigInt | -6.2% | -5.4% | -8.4% | -2.2% | | 5.5% | 28.5% | 15.5% | 33.3% | 37.7% |
| | mixedInt | -6.1% | -5.3% | 1.9% | 0.5% | | 19.8% | 16.0% | 6.4% | 21.7% | 43.9% |
| Mixed | ageTest-1 | -3.2% | -4.3% | -7.8% | 2.3% | | 19.9% | 21.2% | 56.9% | 26.3% | 75.5% |
| | ageTest-2 | -6.5% | -7.3% | -7.5% | -3.9% | | 13.2% | 15.5% | 47.2% | 25.6% | 73.1% |
| | ageTest-3 | -8.3% | -8.5% | -8.4% | -3.4% | | 11.2% | 15.9% | 47.6% | 26.3% | 70.6% |
| NullBool | returnNullBool | -1.4% | 2.6% | -2.3% | -4.9% | | 7.9% | 15.8% | 18.9% | 37.1% | 41.4% |
| | booleanNullAnd-1 | -1.5% | -1.2% | -2.2% | -1.3% | | 14.6% | 14.0% | 33.5% | 33.7% | 46.3% |
| | booleanNullAnd-2 | -1.4% | -1.4% | -1.7% | -2.5% | | 11.6% | 13.1% | 25.8% | 34.4% | 45.8% |
| | booleanNullAnd-3 | -0.9% | -1.4% | -2.7% | 0.4% | | 15.1% | 27.0% | 25.9% | 33.7% | 49.2% |
| | booleanNullAnd-4 | 0.2% | -0.2% | -2.3% | -0.8% | | 11.8% | 15.1% | 20.7% | 34.1% | 46.0% |
| | booleanNullAnd-5 | -2.2% | -0.8% | -0.4% | -4.1% | | 12.1% | 16.5% | 15.4% | 35.9% | 46.7% |
| | booleanNullOr-1 | -1.2% | -0.4% | -1.6% | 0.6% | | 11.4% | 25.2% | 23.5% | 34.7% | 43.7% |
| | booleanNullOr-2 | -0.0% | -0.1% | -2.1% | 1.6% | | 11.2% | 27.3% | 22.8% | 37.9% | 45.9% |
| | booleanNullOr-3 | -2.9% | -3.1% | -3.2% | 0.3% | | 12.9% | 12.0% | 20.5% | 33.3% | 44.5% |
| | booleanNullOr-4 | -1.8% | -1.2% | -1.6% | 1.8% | | 14.3% | 27.5% | 22.9% | 34.5% | 46.4% |
| | booleanNullOr-5 | -1.0% | -1.4% | -1.5% | -2.4% | | 13.6% | 13.4% | 18.5% | 38.0% | 46.6% |
| | booleanNullXor-1 | -4.0% | -4.8% | -3.6% | -1.0% | | 11.5% | 23.3% | 26.8% | 41.0% | 51.0% |
| | booleanNullXor-2 | -4.6% | -5.2% | -4.3% | -0.3% | | 8.5% | 24.2% | 18.5% | 35.1% | 52.6% |
| | booleanNullXor-3 | -3.9% | -3.5% | -3.3% | -0.9% | | 11.3% | 23.8% | 17.8% | 35.5% | 51.1% |
| | booleanNullXor-4 | -3.6% | -3.6% | -3.1% | -0.2% | | 9.9% | 24.8% | 19.7% | 36.2% | 52.3% |
| | booleanNullXor-5 | -4.9% | -4.5% | -3.1% | -2.4% | | 8.9% | 24.7% | 13.1% | 44.8% | 53.7% |
| NullInteger | returnNullInt | 0.6% | 0.9% | -0.8% | -0.7% | | 13.3% | 20.6% | 25.1% | 41.3% | 46.8% |
| | returnNullSmallInt | -2.8% | -1.8% | -2.0% | -2.7% | | 10.0% | 18.1% | 21.6% | 37.6% | 44.5% |
| | returnNullBigInt | -1.4% | 0.3% | -2.0% | -3.9% | | 9.7% | 17.6% | 21.2% | 38.3% | 44.4% |
| | sumNullArgInt-1 | -10.0% | -9.5% | -10.7% | -7.7% | | 1.7% | 18.2% | 12.9% | 24.7% | 44.1% |
| | sumNullArgInt-2 | -8.7% | -8.0% | -10.6% | 4.5% | | 3.9% | 19.6% | 18.6% | 25.4% | 51.6% |
| | sumNullArgInt-3 | -11.7% | -10.4% | -12.5% | 3.5% | | 2.3% | 15.3% | 16.0% | 23.4% | 44.3% |
| | sumNullArgInt-4 | -10.5% | -10.5% | -11.9% | -4.9% | | 22.2% | 17.4% | 19.7% | 25.4% | 50.8% |
| | sumNullArgSmallInt-1 | -9.8% | -9.0% | -10.6% | -7.3% | | 2.5% | 18.6% | 12.2% | 29.2% | 45.6% |
| | sumNullArgSmallInt-2 | -10.0% | -8.7% | -11.0% | 3.9% | | 1.7% | 14.7% | 12.9% | 19.5% | 39.6% |
| | sumNullArgSmallInt-3 | -9.5% | -8.8% | -11.3% | 4.9% | | 6.5% | 13.4% | 14.1% | 19.9% | 39.8% |
| | sumNullArgSmallInt-4 | -8.5% | -9.1% | -10.8% | -3.6% | | 15.5% | 11.2% | 13.8% | 15.5% | 37.4% |
| | sumNullArgBigInt-1 | -9.9% | -8.7% | -10.5% | -7.2% | | 2.8% | 19.2% | 12.2% | 31.5% | 46.7% |
| | sumNullArgBigInt-2 | -9.5% | -8.5% | -10.1% | 4.0% | | 2.4% | 16.9% | 14.9% | 22.1% | 45.5% |
| | checkedSumNullArgInt-1 | -3.3% | -2.5% | -2.5% | -4.3% | | 10.5% | 13.1% | 15.8% | 34.9% | 51.2% |
| | checkedSumNullArgInt-2 | -2.1% | -1.9% | -2.8% | -1.6% | | 11.7% | 13.3% | 22.4% | 38.6% | 55.5% |
| | checkedSumNullArgInt-3 | -2.8% | -1.2% | -1.7% | -0.6% | | 14.4% | 13.2% | 23.8% | 40.8% | 53.4% |
| | checkedSumNullArgInt-4 | -6.0% | -5.8% | -6.3% | 1.3% | | 28.0% | 23.5% | 27.4% | 30.4% | 65.3% |
| | checkedSumNullArgSmallInt-1 | -2.8% | -2.0% | -1.7% | -3.7% | | 12.4% | 13.8% | 17.0% | 36.8% | 57.7% |
| | checkedSumNullArgSmallInt-2 | -1.7% | -1.1% | -1.7% | 0.3% | | 11.2% | 11.7% | 19.0% | 32.2% | 46.8% |
| | checkedSumNullArgSmallInt-3 | -1.6% | -0.7% | -1.6% | 0.3% | | 13.8% | 12.4% | 20.2% | 35.2% | 46.3% |
| | checkedSumNullArgSmallInt-4 | -5.2% | -5.0% | -5.6% | 0.1% | | 19.3% | 15.0% | 18.0% | 19.6% | 44.5% |
| | checkedSumNullArgBigInt-1 | -2.4% | -1.9% | -2.4% | -4.1% | | 11.5% | 13.1% | 16.1% | 35.1% | 50.1% |
| | checkedSumNullArgBigInt-2 | -2.1% | -1.5% | -1.0% | -0.7% | | 12.1% | 12.4% | 20.4% | 35.3% | 52.4% |
| | checkedSumNullArgMixed-1 | -2.8% | -0.3% | -0.7% | -4.6% | | 11.2% | 13.4% | 16.6% | 44.1% | 52.3% |
| | checkedSumNullArgMixed-2 | -2.9% | -2.6% | -1.8% | -0.8% | | 11.8% | 10.7% | 18.5% | 28.2% | 47.8% |
| | checkedSumNullArgMixed-3 | -4.6% | -3.2% | -4.0% | -2.3% | | 12.8% | 11.3% | 20.7% | 33.8% | 55.2% |
| | checkedSumNullArgMixed-4 | -2.4% | -2.1% | -2.0% | -1.6% | | 12.4% | 10.7% | 19.3% | 31.9% | 53.6% |
| | checkedSumNullArgMixed-5 | -6.6% | -5.5% | -5.6% | 1.3% | | 23.1% | 14.2% | 21.0% | 30.4% | 65.3% |
| Recursive | Fibonacci | -19.4% | -19.9% | 114.3% | 3.7% | | 160.2% | 8.9% | -7.7% | 9.0% | 118.1% |
| | Factorial | -12.5% | -16.9% | 56.2% | -2.1% | | 129.6% | 13.9% | -0.3% | 16.1% | 92.5% |
| Results | Best performance | -19.4% | -19.9% | -14.4% | -7.9% | | -4.5% | 6.9% | -7.7% | 9.0% | 30.5% |
| | Worst performance | 0.6% | 5.5% | 114.3% | 20.8% | | 160.2% | 36.7% | 80.9% | 44.8% | 118.1% |
| | Average | -5.2% | -4.7% | -3.5% | -0.9% | | 15.3% | 19.3% | 24.0% | 30.6% | 53.3% |

**Figure 4: Performance heatmap for all PostgreSQL Procedural Languages**

**Table 2: pl/dotnet support types.**

| PostgreSQL | pl/dotnet |
|---|---|
| BIT | `BitArray` |
| BOOL | `bool` |
| BOX | `NpgsqlBox` |
| BPCHAR | `string` |
| BYTEA | `byte[]` |
| CIDR | `IPAddress Address, int Netmask` |
| CIRCLE | `NpgsqlCircle` |
| DATE | `DateOnly` |
| FLOAT4 | `float` |
| FLOAT8 | `double` |
| INET | `IPAddress Address, int Netmask` |
| INT2 | `short` |
| INT4 | `int` |
| INT8 | `long` |
| INTERVAL | `NpgsqlInterval` |
| JSON | `string` |
| LINE | `NpgsqlLine` |
| LSEG | `NpgsqlLSeg` |
| MACADDR | `PhysicalAddress` |
| MACADDR8 | `PhysicalAddress` |
| MONEY | `decimal` |
| PATH | `NpgsqlPath` |
| POINT | `NpgsqlPoint` |
| POLYGON | `NpgsqlPolygon` |
| TEXT | `string` |
| TIME | `TimeOnly` |
| TIMESTAMP | `DateTime` |
| TIMESTAMPTZ | `DateTime` |
| TIMETZ | `DateTimeOffset` |
| UUID | `Guid` |
| VARBIT | `BitArray` |
| VARCHAR | `string` |
| XML | `string` |
| Ranges | |
| DATERANGE | `NpgsqlRange<DateOnly>` |
| INT4RANGE | `NpgsqlRange<int>` |
| INT8RANGE | `NpgsqlRange<long>` |
| TSRANGE | `NpgsqlRange<DateTime>` |
| TSTZRANGE | `NpgsqlRange<DateTime>` |

**Table 3: Tests we wrote for other PSQL Procedural Languages.**

| language | # tests |
|---|---|
| pl/java | 115 |
| pl/lua | 102 |
| pl/perl | 110 |
| pl/pgsql | 109 |
| pl/python | 118 |
| pl/r | 105 |
| pl/tcl | 108 |
| pl/v8(javascript) | 109 |

**Table 4: Comparison of the execution time in relation to pl/csharp.**

| Programming Language | Execution time |
|---|---|
| pl/csharp | 97.87% |
| pl/fsharp | 98.38% |
| pl/pgsql | 100.00% |
| pl/perl | 102.54% |
| pl/python | 103.42% |
| pl/tcl | 119.64% |
| pl/lua | 123.27% |
| pl/java | 128.06% |
| pl/v8 | 134.88% |
| pl/r | 158.75% |

## 6 FUTURE WORK

Work remains after our v1.0 release:

- Datatypes: eight types need to be added: multirange (of which there are five), enumerated, numeric, and composite
- Datatypes: several external PostgreSQL plugins add interesting datatypes, including PostGIS and hstore
- NPGSQL compatiblity will continue improving: improving exception mapping, adding minor SPI features such as subtransactions and notifications, etc.
- More extensive security protections
- More fine-grained control over the runtime environment
- We plan to integrate pl/dotnet more fully with Entity Framework and other parts of the dotnet ecosystem to make development and version management of code across the database boundary seamless and easy

## 7 CONCLUSION

We believe that modern tooling can make stored procedures amazing and return them to the toolbox of software engineers. With pl/dotnet being the fastest and best-tested PL in PostgreSQL, with the only 100compatible database API, the groundwork is in place for us to make this dream into a reality.

We thank the authors of .NET , PostgreSQL, and Npgsql for their work, without which this project would not be possible.

## REFERENCES


[1] Tada AB. 2023. *PL/Java: stored procedures, triggers, and functions for PostgreSQL.* https://tada.github.io/pljava/

[2] Joseph E. Conway. 2023. *PL/R - R Procedural Language for PostgreSQL.* https://github.com/postgres-plr/plr

[3] Microsoft. 2023. *What is .Net?* https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

[4] PostgreSQL. 2022. *PostgreSQL 15 - Documentation.* https://www.postgresql.org/docs/15/intro-whatis.html

[5] RhodiumToad. 2023. *pllua: Embeds Lua into PostgreSQL as a procedural language module.* https://github.com/pllua/pllua


Contents