From pl/v8 → pl/<any>

extending (or discovering) what else a pl/language can do

hannuk@google.com

From pl/v8 → pl/<any>

- Why Server-Side programming
- How **pl/v8** is secretly also **pl/WebAssembly**
- Future: supporting any language in **pl/v8**, turning it into **pl/<any>**

HannuKrosing

Cloud SQL / PostgreSQL

hannuk@google.com

Database functions are good for performance and scalability
Demonstration using **pgbench**

pgbench "<builtin: TPC-B

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)

BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

pgbench "<builtin: TPC-B (sort of)>"
as a **pl/pgsql** function

```
CREATE OR REPLACE FUNCTION pgbench_tpcb_like(
   arg_aid int,
   arg_bid int,
   arg_tid int,
   arg_delta int,
   OUT rbalance int
)
LANGUAGE plpgsql
AS $$
BEGIN
   UPDATE pgbench_accounts SET abalance = abalance + arg_delta WHERE aid =
   SELECT abalance INTO rbalance FROM pgbench_accounts WHERE aid = arg_aid
   UPDATE pgbench_tellers SET tbalance = tbalance + arg_delta WHERE tid =
   UPDATE pgbench_branches SET bbalance = bbalance + arg_delta WHERE bid =
   INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
   VALUES (arg_tid, arg_bid, arg_aid, arg_delta, CURRENT_TIMESTAMP);
END;
$$;
```

pgbench "<builtin: TPC-B (sort of)>"
as a **pl/pgsql** function

Custom pgbench script to use the function

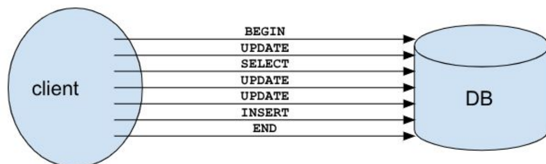
```
cat tpcb-like-plpgsql.pgbench

\set scale 100
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
SELECT 1 FROM pgbench_tpcb_like(:aid, :bid, :tid, :delta);
```

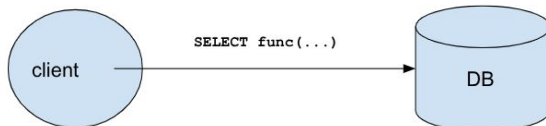
Using the custom pgbench script

```
pgbench -n -T 10 -c 10 pgbench -f tpcb-like-plpgsql.pgbench
```

Database functions are good for performance and scalability Demonstration using **pgbench**

Over 2.5 x better performance even **locally**





Proprietary + Confidential

pgbench plain vs. using a function

```
hannuk:~/work/pgbench-f$ pgbench -n -T 10 -c 16 pgbench
pgbench (16.4 (Debian 16.4-3))
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 100
...
latency average = 1.148 ms
initial connection time = 34.030 ms
tps = 13,940.475167 (without initial connection time)

hannuk:~/work/pgbench-f$ pgbench -n -T 10 -c 16 pgbench
-f tpcb-like-plpgsql.pgbench
pgbench (16.4 (Debian 16.4-3))
transaction type: tpcb-like-plpgsql.pgbench
...
latency average = 0.437 ms
initial connection time = 29.674 ms
tps = 36,653.677985 (without initial connection time)
```

Almost **6 x** when there is lock contention

pgbench plain vs. using a function

```
hannuk:~/work/pgbench-f$ pgbench -n -T 10 -c 64 pgbench10
pgbench (16.4 (Debian 16.4-3))
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
...
latency average = 10.245 ms
initial connection time = 207.649 ms
tps = 6,246.884141 (without initial connection time)

hannuk:~/work/pgbench-f$ pgbench -n -T 10 -c 64 pgbench10
-f tpcb-like-plpgsql.pgbench
pgbench (16.4 (Debian 16.4-3))
transaction type: tpcb-like-plpgsql.pgbench
...
latency average = 1.790 ms
initial connection time = 227.821 ms
tps = 35,760.983670 (without initial connection time)
```

- UDFs are like "normal" programming language functions
- But often it is good to think "microservices"
    - can be called as main entry points to database
    - can have access rights, even locally
    - can be made only ways to interact with database

PostgreSQL is multi-language development platform

- Everything in PostgreSQL is almost infinitely configurable
    - Even most basic things like operator '+' is defined in system tables
    - Also types, casts, index and table access methods, type conversions
    - And most of the time there are functions as part of the definition
- Functions in any language can be called from SQL
- Functions in any language can call each other
- Common data types for arguments and return values are PostgreSQL data types

What is a "pl/v8"

pl/v8 is an extension which exposes Google's V8 javascript engine as pl language in PostgreSQL

```
create or replace function addtwo(vals int[])
returns json as $$
    return vals.map(function(i) {
        return i + 2;
    });
$$ language plv8;

select addtwo(array[0, 47, 30]);
-- Returns [2, 49, 32]
```

https://pgxn.org/dist/plv8/doc/plv8.html

How do PL/ languages work ?

How to create a pl/language

CREATE LANGUAGE

CREATE LANGUAGE — define a new procedural language

Synopsis

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
```

`HANDLER` ***`call_handler`***

`call_handler` is the name of a previously registered function that will be called to execute the procedural language's functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with PostgreSQL as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

How to create a pl/language (2)

Chapter 56. Writing a Procedural Language Handler

The call handler for a procedural language is a "normal" function that **must be written in a compiled language such as C**, using the version-1 interface, and registered with PostgreSQL as taking no arguments and returning the type `language_handler`. This special pseudo-type identifies the function as a call handler and prevents it from being called directly in SQL commands.

> **The call handler is called in the same way as any other function**: It receives a pointer to a `FunctionCallInfoBaseData struct` containing argument values and information about the called function, and it is expected to return a `Datum` result (and possibly set the `isnull` field of the `FunctionCallInfoBaseData` structure, if it wishes to return an SQL null result). *The difference between a call handler and an ordinary callee function is that* ***the `flinfo->fn_oid` field of the `FunctionCallInfoBaseData` structure will contain the OID of the actual function to be called****, not of the call handler itself.* The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target function, not of the call handler.

How to create a pl/language (3)

- Fortunately the part about "must be written in C" is not true
  - Only the function signature is checked in CREATE LANGUAGE
  - Plv8 language handler is called by PostgreSQL even when the handler function provided is written in pl/v8 itself
- This allows to only modify pl/v8 and not core PostgreSQL so that the plv8_language_handler
  - Looks up the custom language actual language handler (called_func→language→handler)
  - and uses that to transpile the provided function source
  - before passing the transpiled code to V8 Javascript Compile function
- There is even an option for the custom language handler to return a compiled Javascript function and not transpiled code, this is used in pl/jsonschema

> **The call handler is called in the same way as any other function**: It receives a pointer to a `FunctionCallInfoBaseData` `struct` containing argument values and information about the called function, and it is expected to return a `Datum` result (and possibly set the `isnull` field of the `FunctionCallInfoBaseData` structure, if it wishes to return an SQL null result). *The difference between a call handler and an ordinary callee function is that* ***the*** ***`flinfo->fn_oid`*** ***field of the*** ***`FunctionCallInfoBaseData`*** ***structure will contain the OID of the actual function to be called****, not of the call handler itself.* The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target function, not of the call handler.

Where we are, what's next ?

Now you **can** write your own pl/language in javascript

https://github.com/plv8/plv8/pull/605

postsql commented on Nov 24, 2025

The "language handler is currently a transpiler from function source to javascript

a sample usage - a language which transpiles any source to "return 42"

language handler which just transpiles any source to "return 42;"

```
CREATE OR REPLACE FUNCTION pseudo_handler()
RETURNS language_handler
LANGUAGE plv8
AS $$
   plv8.elog(NOTICE, "source is [", arguments[0], "]");
   return "return 42";
 $$;
-- CREATE FUNCTION
```

create a language using this handler

```
CREATE OR REPLACE TRUSTED LANGUAGE always42 HANDLER pseudo_handler;
-- CREATE LANGUAGE
```

create a function in this newly created language

```
CREATE OR REPLACE FUNCTION test_always42() RETURNS int LANGUAGE always42 AS $$ return o; $$;
-- CREATE FUNCTION
```

run the function

```
SELECT * FROM test_always42();
NOTICE:  source is [  return o;  ]
 test_always42
---------------
            42
(1 row)
```

postsql commented on Nov 24, 2025 Author

The things that would be also nice to have

- `VALIDATOR` **support** - in case the validator function is written in pl/v8 pass it argument list with types and return type so it can validate that the defined transpiled language can check these
  - I plan to write pl_jsonschema which allows writing JSON Schema validators by wrapping AJV and the validator should allow only a single `JSON` argument and a `boolean` return value
- **Documentation**
- **let the handler return a function**: pl/v8 should recognize when the handler function returns a callable (as AJV does by default) and then use the returned object directly (with currently implemented strict source-to-javascript transpiler architecture I have to extract the function source and have pl/v8 compile it again)

Longer term: WASM support

- **raw arguments/return values**: I already can do WASM support, but it would be more efficient to skip the default argument and return value conversion to Javascript and let WAS handle PostgreSQL data formats directly
  - this would work better if we already compiled core type support tow WASM, like we do for LLM for JIT support
- **support in Rust and other to generate pl/v8 flavour of WASM** - toolkits already package their results for node.js and browsers, they could also support pl/v8