Extending your database server's abilities in a safe and powerful way

PGConf India, 2025
March 05 - 07, 2025

hannuk@google.com

HannuKrosing

Cloud SQL / PostgreSQL

hannuk@google.com

Working with PostgreSQL since it was called `Postgres95` (and also played around with `Postgres 4.2` - without the "SQL" - a little before that).

My oldest *surviving* post on `postgresql-hackers@` mailing list archives is from January 1998, proposing using index for fast ORDER BY queries with LIMIT.

The first DBA at Skype, where I wrote patches for making **VACUUM** able t**o work on more than one table in parallel** and invented the sharding and remote call language **`pl/proxy`** to make it easy to use PostgreSQL in an infinitely scalable way.

Have written books, **PostgreSQL 9 Admin Cookbook** and **PostgreSQL Server Programming**

After Skype I did 10+ years of PostgreSQL consulting all over the world as part of 2ndQuadrant.

For last six years he has been a PostgreSQL Database Engineer at Google working mostly with PostgreSQL / Cloud SQL.

Plan for this talk

- What is a "Database Server" and why Server-Side programming
- What is a "pl/" language, types and flavours of PgSQL languages
- How **pl/v8** is secretly also **pl/WebAssembly**
- Future: supporting any language in **pl/v8**, turning it into **pl/<any>**

What is a Database Server (and why Server-Side programming)

- Database server is **not** primarily "a place to store data"
- It is about organization, correctness and and control
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Interacting with Database Server

- You send you code (processing requests) to the data
- You get back result, usually some data, or status
- SQL is not complicated
-
-

Interacting with Database Server

- You send you code (processing requests) to the data
- You get back result, usually some data, or status
- SQL is not complicated
- … but data management is
- SQL is just a representation of this complexity

SQL is a "no-code development environment"

- You tell the server in "an English-like language" what you want to do
- And it figures out all the complexities of doing it efficiently and securely
-
-
-

SQL is a "no-code development environment"

- You tell the server in "an English-like language" what you want to do
- And it figures out all the complexities of doing it efficiently and securely
- If you want to sound fancy then you can call yourself a SQL Prompt Engineer
- The difference from AI / LLM is that the results are predictable and guaranteed once you master SQL. No need to play with "temperature" to get good results

Why use database function

- Some claim "Business logic should not be in database !"
- In fact modern Relational Database IS the place to enforce business logic
- Only place that can guarantee data consistency
- Doing it anywhere else is lot's of work and almost alway has gaps
- Database functions extend this to units of work.
- Think of your database functions as **Microservices** !

Database functions are also good for performance and scalability

Demonstration using **pgbench**

pgbench "<builtin: TPC-B (sort of)>"

```
\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 $$
BEGIN
   UPDATE pgbench_accounts SET abalance = abalance + arg_delta WHERE aid = arg_aid;
   SELECT abalance INTO rbalance FROM pgbench_accounts WHERE aid = arg_aid;
   UPDATE pgbench_tellers SET tbalance = tbalance + arg_delta WHERE tid = arg_tid;
   UPDATE pgbench_branches SET bbalance = bbalance + arg_delta WHERE bid = arg_bid;
   INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
   VALUES (arg_tid, arg_bid, arg_aid, arg_delta, CURRENT_TIMESTAMP);
END;
$$;
```

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

Custom pgbench script to use the function

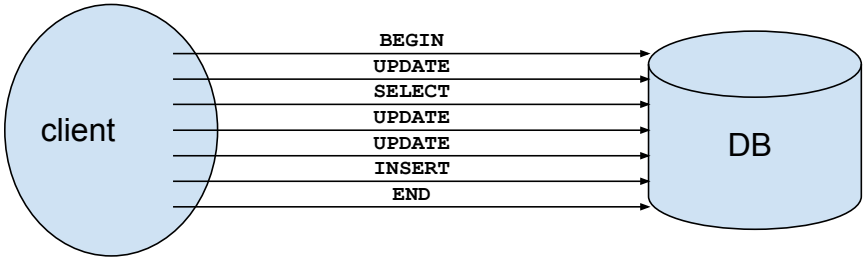
```
cat tpcb-like-plpgsql.pgbench

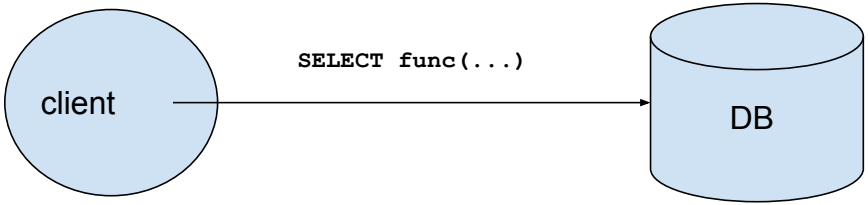
\set scale 100
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
SELECT 1 FROM pgbench_tpcb_like(:aid, :bid, :tid, :delta);
```

Using the custom pgbench script

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

Over 2.5 x better performance



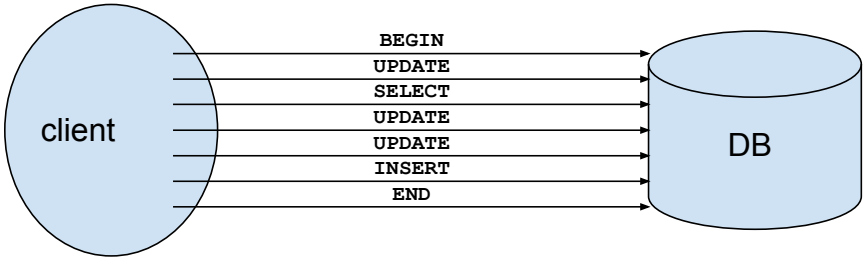


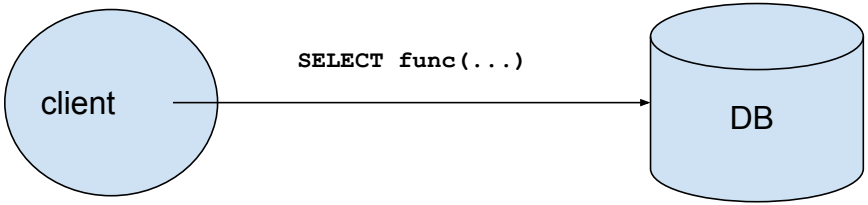
pgbench plain vs. using a function

```
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** with 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)
```

PostgreSQL as a 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

PostgreSQL has many types of Functions

- simplest is scalar function - **`abs(-7)` → `7`**
- Functions can also return tuples (though this is still "kind of scalar")
- And they can RETURN TABLE (same as "SETOF record")
- They can operate on sets of records and return scalars (AGGREGATE and WINDOW)
- They can implement TRIGGERs (RETURNS TRIGGER)
- Some are "internal", some are undocumented and can change

Functions in PostgreSQL by "language type"

- "C" and "internal"
- "pl/..." - interpreted languages

```
CREATE OR REPLACE FUNCTION add_abs(a int, b int, OUT c int)
LANGUAGE plpgsql
AS 'BEGIN c := abs(a) + abs(b); END';

SELECT add_abs(3, -7);
 add_abs
---------
      10
(1 row)
```

```
CREATE OR REPLACE FUNCTION add_abs(a int, b int, OUT c int)
LANGUAGE plpgsql
AS $$
BEGIN
    c := abs(a) + abs(b);
END';
$$;
```

- SQL and "ANSI/ISO standard SQL"

```
CREATE FUNCTION(…)… BEGIN ATOMIC … END;
```

Trusted and Untrusted languages

- C and internal are always all-powerful, that is "untrusted"
- Untrusted - pl/perl**u**, pl/python**u**, pl/R ← *no "u", still untrusted*
- SQL is always trusted
- Trusted - i.e. nicely sandboxed - pl/perl, pl/tcl, **pl/v8**

What is a "V8"

- V8 is a
  - JIT compiled **Javascript** interpreter
  - for Google Chrome,
  - by Google
- High performance, Stable, keeps up with latest standards
- Runs **`JavaScript`** and **`wasm`**
- Lot's of work on security and sandboxing, process isolation, etc
- Only a subset of this security work is applicable to pl/v8
- No versioned / stable libv8.so released by Google Chrome team
  - when this was announced Linuxes dropped pl/v8 packaging

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

Getting pl/v8

- When you use any of the Cloud providers pl/v8 is just there

```
plv8db=> CREATE EXTENSION plv8;
CREATE EXTENSION
Time: 596.566 ms
```

- Linux distribution do not currently have it
- But it is easy to add
  - Get the source from https://github.com/plv8/plv8
  - Make sure you have all the **Build Requirements** as describe in README
  - Run `make`
  - Run `sudo make install`

- There is some hope that it will be packaged for Linux in the future

Verifying JSON

Verifying JSON - against JSON Schema

- PostgreSQL has support for JSON data type
- JSON itself is defined as anything goes
- There is no built-in JSON structure validation in PostgreSQL
- Existing standard for describing JSON structure is JSON Schema
- There exist many implementations of JSON Schema validation

Option 1 - pg_jsonschema

- Written in Rust
- Uses PGRX to wrap a Rust module
- Compiles to native code
-

pg_jsonschema

postgresql 12+ license MIT

Source Code: https://github.com/supabase/pg_jsonschema

Summary

`pg_jsonschema` is a PostgreSQL extension adding support for JSON schema validation on `json` and `jsonb` data types.

API

Three SQL functions:

- json_matches_schema
- jsonb_matches_schema (note the **jsonb** in front)
- jsonschema_is_valid

With the following signatures

```
-- Validates a json *instance* against a *schema*
json_matches_schema(schema json, instance json) returns bool
```

Option 1 - pg_jsonschema

- Written in Rust
- Uses PGRX to wrap a Rust module
- Compiles to native code
- Validaton function, can be used in **CHECK** constraint

```
-- Validates a jsonb *instance* against a *schema*
jsonb_matches_schema(schema json, instance jsonb) returns bool
```

```
create extension pg_jsonschema;

create table customer(
    id serial primary key,
    metadata json,

    check (
        json_matches_schema(
            '{
                "type": "object",
                "properties": {
                    "tags": {
                        "type": "array",
                        "items": {
                            "type": "string",
                            "maxLength": 16
                        }
                    }
                }
            }',
            metadata
        )
    )
);

-- Example: Valid Payload
insert into customer(metadata)
values ('{"tags": ["vip", "darkmode-ui"]}');
-- Result:
--   INSERT 0 1
```

Option 2 - use pl/v8

- Javascript has many packages for validation

  JSON

- https://github.com/ajv-validator/ajv is one

  of the fastest ones

-

README | Code of conduct | MIT license | Security

Ajv JSON schema validator

The fastest JSON validator for Node.js and browser.

Supports JSON Schema draft-04/06/07/2019-09/2020-12 (draft-04 support requires ajv-draft-04 package) and JSON Type Definition RFC8927.

Prepare AJV for pl/v8

- Start from npm package
- Use browserify to combine the whole package int a single javascript file
- Now you have code which can be run to add avj to your plv8 environment
- The **`ajv_bundle.sql`** file is about **230 KB**

Configuration

$ nvm install 16 /* Install the desired version of Node.js */
$ npm install ajv
$ npm install -g browserify
$ vi ajv_utils.js /* Add the following */

```
// Node.js require:
const Ajv = require("ajv")

ajv_class = function ajv_class() {
const ajv = new Ajv()
return ajv;
}
```

/* Create a unique javascript bundle file; it could be interesting to try plv8ify */
$ browserify ajv_utils.js -o ajv_bundle.js
$ cat >ajv_bundle.sql << EOF
create or replace function ajv() returns void as \$\$
$(<ajv_bundle.js)
\$\$ language plv8;
EOF

Setup function for AJV for pl/v8

```
plv8db=> CREATE OR REPLACE FUNCTION plv8_init() RETURNS void LANGUAGE plv8
AS $plv8$
   /* Calling top-level function ajv() sets up the ajv
      jsonschema validator environment */
   const ajv_setup = plv8.find_function("ajv");
   ajv_setup();
  /* Do here as many as possible common and "heavy" ops and share results
     with other plv8 functions via the plv8 object properties */
   plv8.ajv = ajv_class();
   plv8.validator = {};
$plv8$;

/* Load the "plv8_init" procedure at startup; make sure the setup is
  done before any plv8 function is called */
plv8db=> ALTER DATABASE pg_jsonschema SET plv8.start_proc = 'plv8_init';
```

Using AJV for pl/v8: the validator function

```
/* Create the function to be later called in the DDL create table check constraint */
plv8db=> CREATE OR REPLACE FUNCTION plv8_json_matches_schema(
 schema text, data jsonb, tablename text
) returns boolean as
$$
validator = plv8.validator[tablename];
if (!validator) {
       validator = plv8.ajv.compile(JSON.parse(schema));
       plv8.validator[tablename] = validator;
       }
return validator(data);
$$
language plv8;
```

Using AJV for pl/v8: using the validator

```
plv8db=> create table <schema>.pgbench_test_pg_jsonschema_a(
   meta jsonb,
   CHECK (
       plv8_json_matches_schema(
           '{"type": "object", "properties": {"a": {"type": "number"}, "b": {"type": "string"}}}',
           meta, '<schema>.bench_test_pg_jsonschema_a'
       )
   )
);

Postgres=> insert into bench_test_pg_jsonschema_a(meta) values ('{"a": 4, "b": "7"}');
-- INSERT 0 1
Postgres=> insert into bench_test_pg_jsonschema_a(meta) values ('{"a": "4", "b": "7"}');
-- ERROR:  new row for relation "bench_test_pg_jsonschema_a" violates check constraint
"bench_test_pg_jsonschema_a_meta_check"
-- DETAIL:  Failing row contains ({"a": "4", "b": "7"}).
```

Validation the performance

AVJ in pl/v8

```
plv8db=> insert into bench_test_pg_jsonschema_a(meta)
select
   json_build_object(
       'a', i,
       'b', i::text
   )
from
   generate_series(1, 20000) t(i);

-- First run with one-time schema compilation
-- INSERT 0 20000
-- Time: 341.130 ms
-- Second run
-- INSERT 0 20000
-- Time: 255.099 ms
-- Third run, faster still V8 optimises in the background
-- INSERT 0 20000
-- Time: 243.636 ms
```

pg_jsonschema

```
create table bench_test_pg_jsonschema(
   meta jsonb,
   check (
       jsonb_matches_schema(
           '{"type": "object", "properties": {"a": {"type":
"number"}, "b": {"type": "string"}}}',
           meta
       )
   )
);

insert into bench_test_pg_jsonschema(meta)
select
   json_build_object(
       'a', i,
       'b', i::text
   )
from
   generate_series(1, 20000) t(i);
-- Query Completed in 351 ms
```

Adding a WebAssembly function

Adding a new hash function - sha3-256

- Sha3-256 is a newer hash function which is considered better than sha256 but is also more CPU intensive
- It is available in pgcrypto extension from contrib/
- For this demonstration we use a WebAssembly implementation from https://github.com/Daninet/hash-wasm
- We just install it in node: `npm i hash-wasm`, then extract the module
- Now the bas64-encoded WASM module is inside `~/node_modules/hash-wasm/dist/sha3.umd.min.js`

Using WASM: make module source available as **bytea**

```
-- wrap the WASM code for easy usage
CREATE FUNCTION wasm.code_hash_wasm_sha3(OUT code bytea)
LANGUAGE SQL
BEGIN ATOMIC
 SELECT decode($wasm$
AGFzbQEAAAABFARgAAF/YAF/AGACf38AYAN/f38AAwgHAAEBAgEAAwUEAQECAgYOAn8BQZCNBQt/AEGACAsHcAgGbWVtb3J5AgAOSGFzaF9HZXRCdWZmZXIAAAlIYXNoX0luaXQAAQtIYXNoX1VwZGF0
ZQACCkhhc2hfRmluYWwABA1IYXNoX0dldFN0YXRlAAUOSGFzaF9DYWxjdWxhdGUABgpTVEFURV9TSVpFAwEKpBwHBQBBgAoL1wMAQQBCADcDgI0BQQBCADcD+IwBQQBCADcD8IwBQQBCADcD6IwBQQBC
ADcD4IwBQQBCADcD2IwBQQBCADcD0IwBQQBCADcDyIwBQQBCADcDwIwBQQBCADcDuIwBQQBCADcDsIwBQQBCADcDqIwBQQBCADcDoIwBQQBCADcDmIwBQQBCADcDkIwBQQBCADcDiIwBQQBCADcDgIwB
QQBCADcD+IsBQQBCADcD8IsBQQBCADcD6IsBQQBCADcD4IsBQQBCADcD2IsBQQBCADcD0IsBQQBCADcDyIsBQQBCADcDwIsBQQBCADcDuIsBQQBCADcDsIsBQQBCADcDqIsBQQBCADcDoIsBQQBCADcD
mIsBQQBCADcDkIsBQQBCADcDiIsBQQBCADcDgIsBQQBCADcD+IoBQQBCADcD8IoBQQBCADcD6IoBQQBCADcD4IoBQQBCADcD2IoBQQBCADcD0IoBQQBCADcDyIoBQQBCADcDwIoBQQBCADcDuIoBQQBC
...

aiABQYCKAWooAgA2AgAgAUGECmogAUGEigFqKAIANgIAIAFBiApqIAFBiIoBaigCADYCACABQYwKaiABQYyKAWooAgA2AgAgAUEQaiEBIAAgBEEEaiIERw0ACwsgBUUNACAFQQJ0IQAgBEECdCEBA0Ag
AUGACmogAUGAigFqKAIANgIAIAFBBGohASAAQXxqIgANAAsLCwvYAQEAQYAIC9ABkAEAAAAAAAAAAAAAAAAAAAAEAAAAAAAAAgoAAAAAAAAACKgAAAAAAAgACAAIAAAACAi4AAAAAAAAAABAACAAAAAAAIGA
AIAAAACACYAAAAAAAICKAAAAAAAAAIgAAAAAAAAACYAAgAAAAAAKAACAAAAAAIuAAIAAAAAAiwAAAAAAAICJgAAAAAAAgAOAAAAAAAACAAoAAAAAAAICAAAAAAAAAgAqAAAAAAAAACgAAgAAAAICBgACA
AAAAgICAAAAAAAACAAQAAgAAAAAAIgACAAAAAAgA==
$wasm$, 'base64');
END;
```

```
plv8wdb=# select length(wasm.code_hash_wasm_sha3());
┌────────┐
│ length │
├────────┤
│   4018 │
└────────┘
(1 row)
```

Using WASM: compile the function, set up environment

```
-- set up sha3 wasm code and calling API
CREATE OR REPLACE FUNCTION wasm.plv8_prepare_sha3(code bytea)
RETURNS void
LANGUAGE plv8
IMMUTABLE STRICT
AS $plv8$
   /* sync init of WASM code */
   const mod = new WebAssembly.Module(code);
   const instance = new WebAssembly.Instance(mod, {});
   const chunksize = 16384;
   const arrayOffset = instance.exports.Hash_GetBuffer()
       , memoryBuffer = instance.exports.memory.buffer;
   /* the memryView is used to pass data to and from WASM */
   const memoryView = new Uint8Array(memoryBuffer, arrayOffset, chunksize);

   plv8.SHA3_Interface = {
       Hash_Init:  instance.exports.Hash_Init,
       Hash_Update: instance.exports.Hash_Update,
       Hash_Final:  instance.exports.Hash_Final,
       memoryView: memoryView
   }
$plv8$;
```

Using WASM: the function to compute sha3 hash

```
-- the actual function
CREATE OR REPLACE FUNCTION public.hash_sha3_256(data bytea, out hash_sha3 bytea)
LANGUAGE plv8
IMMUTABLE STRICT
AS $plv8$
   const chunksize = 16384;
   if ("undefined" == typeof plv8.SHA3_Interface)
       plv8.execute("SELECT wasm.plv8_prepare_sha3(wasm.code_hash_wasm_sha3());");
   SHA3X = plv8.SHA3_Interface;

   SHA3X.Hash_Init(256)
   let read = 0;
   while (read < data.length) {
     const chunk = data.subarray(read, read + chunksize);
     read += chunk.length;
     SHA3X.memoryView.set(chunk);
     SHA3X.Hash_Update(chunk.length);
   }
   SHA3X.Hash_Final(0x06)
   return SHA3X.memoryView.slice(0, 256 / 8);
$plv8$;
```

Testing performance

Hashing 1 million short strings - wasm 5.4 sec, native 1.4, difference 3.8 x

```
plv8wdb=# SELECT i FROM generate_series(1, 1000000) i WHERE LENGTH(hash_sha3_256(('foobar'||i)::BYTEA))>32;
...
Time: 5374.943 ms (00:05.375)
plv8wdb=# SELECT i FROM generate_series(1, 1000000) i WHERE LENGTH(digest(('foobar'||i)::BYTEA, 'sha3-256'))>32;
...
Time: 1408.928 ms (00:01.409)
```

Hashing 1 million 1188-byte strings - wasm 12.9 sec, native 6.2, difference 2.1 x

```
plv8wdb=# SELECT i FROM generate_series(1, 1000000) i WHERE LENGTH(hash_sha3_256((repeat('foobar'||i,
100))::BYTEA))>32;
...
Time: 12894.676 ms (00:12.895)
plv8wdb=# SELECT i FROM generate_series(1, 1000000) i WHERE LENGTH(digest((repeat('foobar'||i, 100))::BYTEA,
'sha3-256'))>32;
...
Time: 6201.708 ms (00:06.202)
```

What's next ?

Where from here

- Reducing complexity for WASM -
  - Generate **pl/v8 wasm** function automatically when compiling
  - Call specification inside wasm module ?
- Reducing complexity for using AVJ / JSONSchema
  - Instead of setup function create language
  - Write a language_handler in pl/v8 (we are making good progress on this!)
  - `CREATE LANGUAGE pg_jsonschema HANDLER …;`
  - JSON Schema becomes the source code for validator function!
  - ```
    CREATE OR REPLACE FUNCTION plv8_json_matches_schema(OUT boolean)
    LANGUAGE plv8 AS $$ <JSON SCHEMA HERE>$$;
    ```
- Allow any other javascript transpiled language


    ```

Questions ?

Hannu Krosing

Cloud SQL / PostgreSQL

hannuk@google.com