Writing Safe Postgres extensions with Rust

DALIBO
L'expertise PostgreSQL

Damien Clochard

- Co-founder of DALIBO, leading Postgres company in France since 2005
- Active member of the French PostgreSQL community
- Main developer of the PostgreSQL Anonymizer extension

My journey

I discovered Postgres 25 years ago

I discovered Rust last year

My Story

In 2018, I started a project called PostgreSQL Anonymizer

Over the years, I wrote more and more C code…

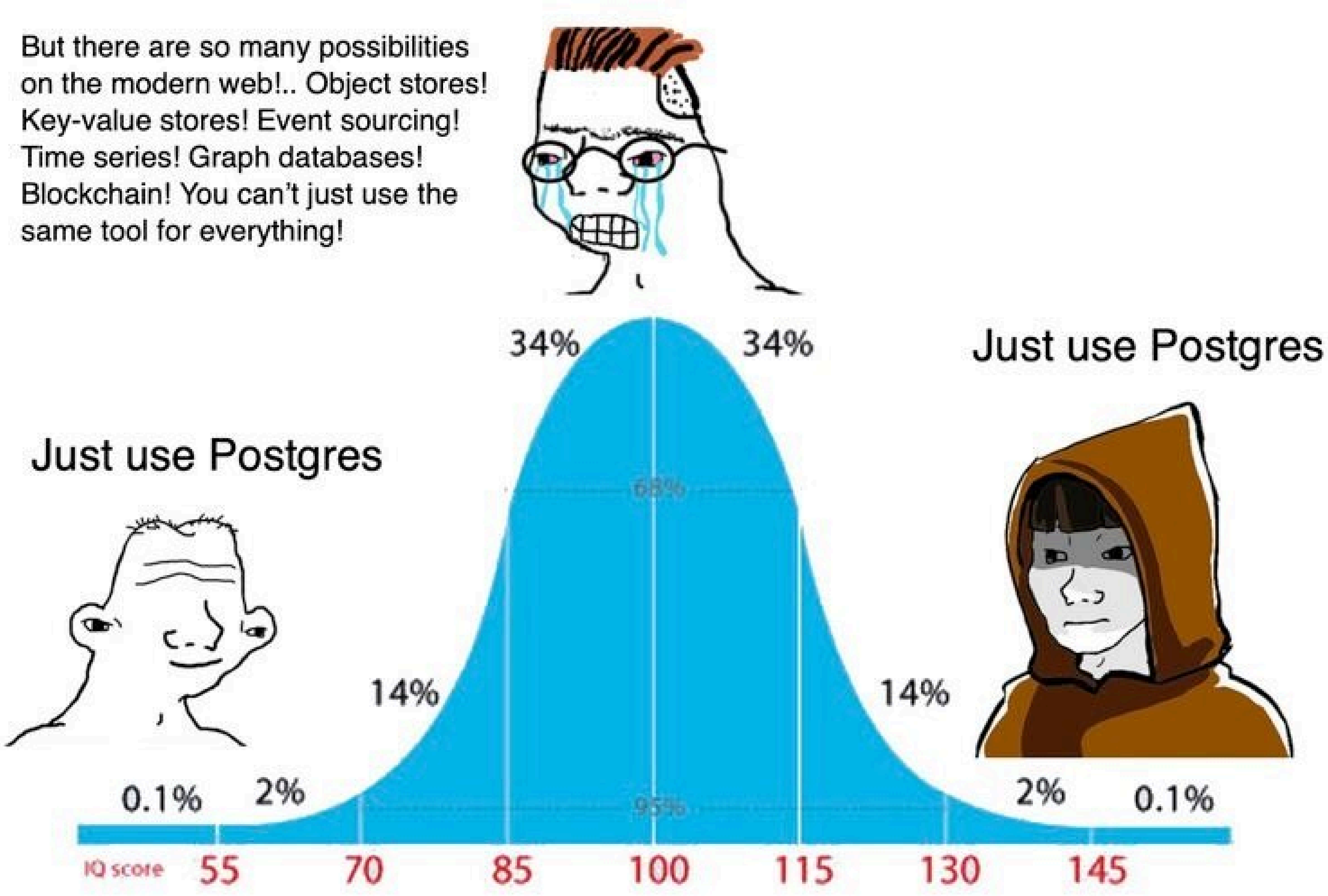
Last year, I rewrote everything in Rust

This is my story :)

Menu

- What are Postgres Extensions ?
- The PGRX Framework
- A practical example
- Lessons learned from Postgres Anonymizer 2.0
- Postgres future is rusty !

Just use Postgres.


But there are so many possibilities on the modern web!.. Object stores! Key-value stores! Event sourcing! Time series! Graph databases! Blockchain! You can't just use the same tool for everything!
Just use Postgres
Just use Postgres
34%
34%
68%
14%
14%
95%
0.1%
2%
2%
0.1%
IQ score
55
70
85
100
115
130
145

DALIBO
L'expertise PostgreSQL

Postgres is not a database, it’s a platform

- **Graphs** ? Apache AGE, EdgeDB
- **Geo Data** ? PostGIS, pg_pointcloud, pg_routing
- **OLAP** ? pg_DuckDB, Citus, Hydra, pg_analytics
- **NoSQL** ? JSONB, FerretDB
- **AI** ? pgvector, PostgresML
- **ETL** ? 100+ Foreign Data Wrappers
- **Timeseries** ? TimescaleDB
- **Full Text Search** ? pgroonga, ParadeDB, zombodb
- **API** ? pg_graphql, postgREST
- **Pluggable storage** ? OrioleDB, Neon
- etc….

A unique ecosystem

More than 1000 known extensions

… almost 250 are active and maintained

So what is an Postgres extension ?

- Some SQL objects
- and/or Procedural Language (PL) code
- and/or a compiled library

Writing an extension in SQL or PL code

- Easy and Fast
- A great way to share code between several databases
- very stable between major versions
- 20 procedural languages (PL)

```sql
CREATE OR REPLACE FUNCTION get_employee_name(emp_id INTEGER)
RETURNS VARCHAR AS $$
DECLARE
    emp_name VARCHAR;
BEGIN
    SELECT first_name || ' ' || last_name INTO emp_name
    FROM employees WHERE id = emp_id;
    RETURN emp_name;
END;
$$ LANGUAGE SQL;
```

```sql
SELECT get_employee_name(123);
```

20+ procedural languages

PL/pgsql	PL/perl	PL/php	PL/Ruby	PL/Java
PL/Scheme	PL/tcl	PL/Lua	PL/python	PL/haskell
PL/Rust	PL/dotnet	PL/lolcode	PL/Julia	PL/sh
PL/XSLT	PL/R	PL/v8	PL/go	PL/brainfuck

PL/Rust ?

- We’re not going to talk about it today :)

Writing an extension C

- Generaly loaded via `shared_preload_libraries`
- Direct access to the internal functions
- Absolutely no security barrier => segfaults fest
- Very low-level code / No Abstractions
- Each new major version will probably break your extension
- The dev/test framework (PGXS) is *very* limited
- if the extension crashes, the entire Postgres instance will crash too

if the extension crashes,

the entire Postgres instance will crash too

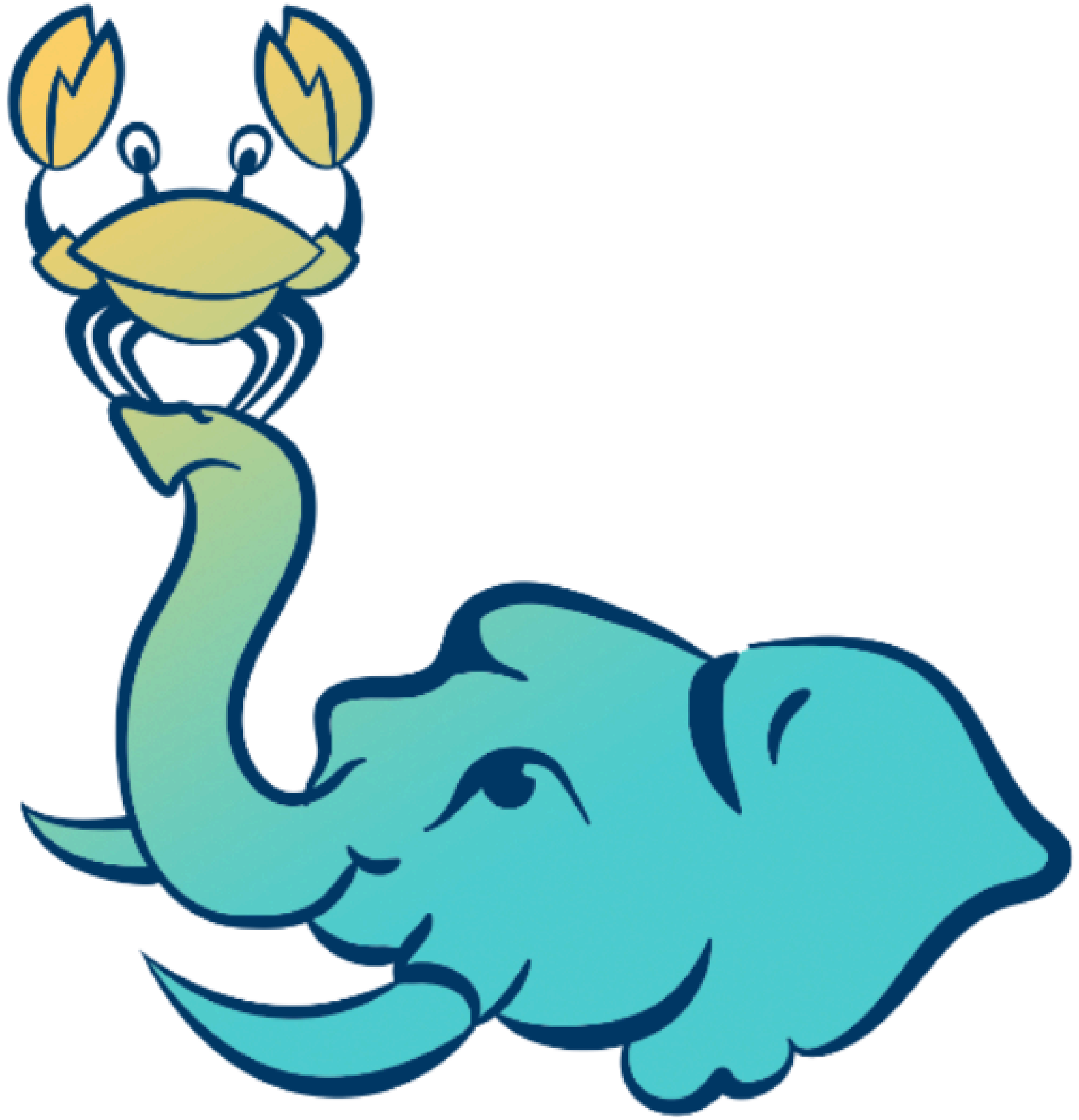
Can we have the best of both languages ?

Safety of PL functions **AND** Performances of C

High Level Abstractions **AND** Access to Postgres internals

A modern language **AND** Stability

PGRX : a breath of fresh air



PGRX : playing with crabs and elephants

A framework that bridges the gap between rust and postgres

PGRX : principles

- Expose your rust functions as user functions inside postgres
- Automatic mapping postgres data types into rust types
- … and vice versa

PGRX : type conversions

postgres	`rust`
BYTEA	`Vec<u8>` or `&u8`
TEXT	`String` or `&str`
INTEGER	`i32`
DATE	`pgrx::Date`
DATERANGE	`pgrx::Range<pgrx::Date>`
NULL	`Option::None`

PGRX : A bridge between 2 worlds

- Derive attributes and macros to export Rust functions in SQL
- Rust abstractions over Postgres pointers (`pgBox<T>`)
- Helpers to exchange memory with Postgres
- Safe access to the Server Programming Interface (SPI)
- Any Rust `panic!` is translated into a Postgres ERROR

PGRX : Safety Fist

If the extension crashes

An ERROR event is raised into Postgres

The transaction is cancelled (ROLLBACK)

The current session lives on

The Postgres instances survives

PGRX : Modern development tooling

- A fully managed development environment (`cargo-pgrx`)
- All major versions are supported
- An idiomatic Rust Test framework
- A very nice development feedback loop
- Easy commands to build and ship packages

PGRX : an open and active community

- Project launched by a single company (`TCDI`)
- Transfered last year to the pgcentral foundation
- A friendly Discord channel for beginners

A practical example

Let’s go …

```
cargo install --locked cargo-pgrx
cargo pgrx init
```

… to a new world !

```
cargo pgrx new world
cd world
```

src/lib.rs

```rust
#[pg_extern]
fn hello_world() -> &'static str {
    "Hello, world"
}
```

Let’s try this !

```
cargo pgrx run
```

Bonjour tout le monde !

```
world=# CREATE EXTENSION world;

world=# SELECT hello_world();
 hello_world
--------------
 Hello, world
(1 row)
```

Add a parameter

```rust
#[pg_extern]
fn hello(name: &str) -> String {
    format!("Hello, {name}")
}
```

Bonjour FOSDEM !

```
world=# DROP EXTENSION world;
world=# CREATE EXTENSION world;
```

```
world=# SELECT hello('FOSDEM');
   hello
------------
 Hello, FOSDEM
(1 row)
```

Same function with C ?

```c
PG_FUNCTION_INFO_V1( hello );
Datum hello( PG_FUNCTION_ARGS ) {
   char hello[] = "Hello, ";
   text * name;
   int hellolen;
   int namelen;
   text * msg;
   name = PG_GETARG_TEXT_P(0);
   hellolen = strlen(hello);
   namelen = VARSIZE(name) - VARHDRSZ;
   msg = (text *)palloc( hellolen + namelen );
   SET_VARSIZE(hello, hellolen + namelen  + VARHDRSZ );
   strncpy( VARDATA(msg), hello, hellolen );
   strncpy( VARDATA(msg) + hellolen, VARDATA(name), namelen );
   PG_RETURN_TEXT_P( msg );
}
```

Any Canadians in the room ?

The Canadian Social Insurance Number (SIN) is composed of 8 digits + 1 control digit

```
046 454 286
```

The control digit is computed using the Luhn Formula

```
luhn("046 454 28") = 6
```

Do not reinvent the wheel

```
cargo add luhn3
```

Compute the checksum

```
#[pg_extern]
fn luhn_checksum(input: &str) -> char {
    use luhn3::decimal::checksum;
    checksum(input.as_bytes())
        .expect("Input should be decimal")
        as char;
}
```

Rebuild the extension

```
cargo pgrx run
```

Manual testing

```
SELECT luhn_checksum('04645428');
 luhn_checksum
---------------
 6
```

```
SELECT luhn_checksum('A');
ERROR:  Input should be decimal
```

Automatic testing

```
#[cfg(any(test, feature = "pg_test"))]
#[pg_schema]
mod tests {
    use pgrx::prelude::*;

    #[pg_test]
    fn test_luhn_checksum() {
        assert_eq!("8", crate::luhn_checksum("1"));
    }
}
```

Launch the tests

```
cargo pgrx test
[...]
test tests::pg_test_luhn_checksum ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out;
finished in 5.26s
```

Launch tests on Postgres 14

```
cargo pgrx test pg14
```

Create a User Defined Type (UDT)

```
#[derive(PostgresType, Serialize, Deserialize, Debug)]
#[inoutfuncs]
pub struct SIN (i32);
```

Implement basic I/O traits

```rust
impl InOutFuncs for SIN {

  fn input(input: & core::ffi:CStr) -> Self { ... }

  fn output(&self, buffer: &mut pgrx::StringInfo) { ... }

}
```

Input trait

```rust
fn input(input: & core::ffi:CStr) -> Self {
  use luhn3::decimal::valid;
  let val = input.to_str().expect("Invalid Input").replace(' ', "");
  if ! valid(&val.clone().into_bytes()) {
    error!("{}", "Not a valid SIN");
  }
  SIN(val.parse::<i32>().expect("Value should be a number") / 10)
}
```

Output trait

```
fn output(&self, buffer: &mut pgrx::StringInfo) {
  use luhn3::decimal::checksum;
  let part1 = self.0 / 100000 % 1000;
  let part2 = self.0 / 100 % 1000;
  let part3 = self.0 % 100;
  let part4 = checksum(&self.0.to_string().into_bytes())
      .expect("Checksum Failed")
      as char;
  let val = format!("{part1:03} {part2:03} {part3:02}{part4}");
  buffer.push_str(val.as_str());
}
```

Let’s try that new type

```
cargo pgrx run
```

Convert a TEXT into SIN

```
SELECT CAST ( '046454286' AS SIN );
    SIN
-------------
 046 454 286
```

```
SELECT CAST ('999 999 999' AS SIN);

ERROR:  Not a valid SIN
```

Use this type in a column

```
CREATE TABLE canadians (id SIN PRIMARY KEY, name TEXT);

ERROR:  data type sin has no default operator class for access
        method "btree"
HINT:   You must specify an operator class for the index or define
        a default operator class for the data type.
```

Derive the default operators

```
#[derive(PostgresType, Serialize, Deserialize, Debug, PartialEq)]
#[derive(Eq, PartialEq, Ord, Hash, PartialOrd)]
#[derive(PostgresEq)]
#[derive(PostgresOrd)]
#[inoutfuncs]
pub struct SIN(i32);
```

Compare 2 SINs

```
SELECT '483247862'::SIN > '483247870'::SIN ;

 ?column?
----------
 f
(1 row)
```

Now the column works

```
CREATE TABLE canadians (id SIREN PRIMARY KEY, name TEXT);

INSERT INTO canadians VALUES ('483247862','James Howlett');

SELECT * FROM canadians;
     id      |  name
-------------+---------------
 483 247 862 | James Howlett
```

There’s so much more….

You can also interact with the database engine itself !

- Foreign data wrappers
- 30+ Hooks
- WAL decoders for logical replication
- Index Access Methods
- Table Access Methods

Feedback

From C to Rust

Rewriting PostgreSQL Anonymizer from scratch

- A data masking extension for PostgreSQL
- I’ll talk about it tomorow at 15h00 in the Postgres devroom (UA2.220)
- About 1000 lines of C code
- Rewrote everything in a few weeks, without prior knowledge of Rust

A feeling of « déja vu »

There's some unspoken familiarity between Postgres and Rust

- The Rust compiler is dull and rough at the beginning
- But once you climbed that learning curse, you're rewarded
- …. pretty much like Postgres :)

Immediate Gains

- Confort of development
- Dozens of unit tests => many bugs found along the way
- Better performance by rewriting some PL/pgSQL in Rust
- Using high level Rust crates (`faker-rs`, `image`)
- Stability (« no more segfaults ! »)

Culture Shock

- In Rust a variable is never `NULL` !
- Some Postgres internal macros are missing
- Handling 2 memory contexts at once
- Some bindings are still missing

It’s not magic

- A lot of sections in the code are still `unsafe`
- You still need to read and understand the Postgres C code
- Building is very very slow (about 20x slower than C)
- No support de Windows at the moment

The Postgres Future is Rusty !

A new generation of extensions

- supabase wrappers
- PL/PRQL
- Timescaledb-toolkit
- pg_graphql
- pgvecto.rs
- pg_later
- paradeDB
- pgmq
- neon
- pgzx

Join the revolution

- Bring back your code close to the data
- Define your own types !
- Use Postgres as a platform
- Rust extensions are a great entrypoint to the Postgres community

Links

PGRX

https://github.com/pgcentralfoundation/pgrx

A 4 hour tutorial

https://daamien.gitlab.io/pgrx-tuto/

Try out PostgreSQL Anonymizer !

https://gitlab.com/dalibo/postgresql_anonymizer

MERCI !