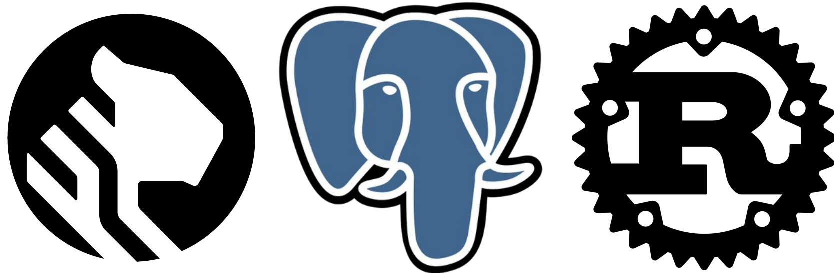


Fearless Extension Development with **Rust** and **PGRX**

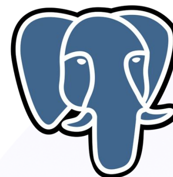
James Blackwood-Sewell

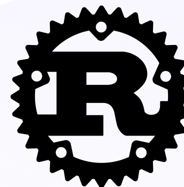
"I can't believe it's not C"

James
Blackwood-Sewell

Lives in

Loves to

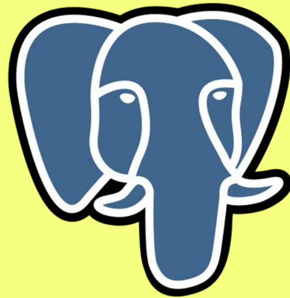




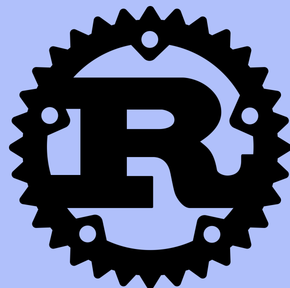
Works at

Timescale

You all know what Postgres is



But what about Rust?



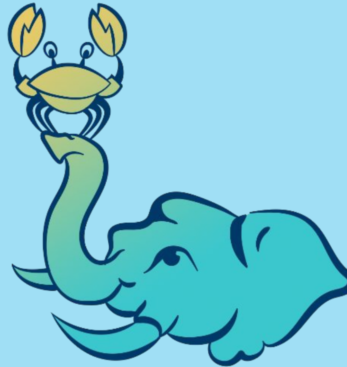
Rust is best described by its primary features

1. Modern Compiled Language
2. Safety by Construction
3. Low Level Control
4. Compatibility with C
5. Tooling (pkgs, build, testing)
6. Can run inside Postgres

* important for me

Great slides: Considering Rust …

And what about PGRX?



PGRX is a Framework for PostgreSQL

- PGRX exposes the PostgreSQL C API via safe Rust code, removing many opportunities for crash or corruption
- PGRX Rust code is compiled and runs close to the speed of C, and many times faster than code in a PL/ language
- PGRX helps out with your development process; from auto-creating SQL objects, to testing and packaging your extension
- PGRX makes high performance PostgreSQL Extensions more accessible

A bit of history …

* things got good around here

Benjamin Fry starts pg-extend-rs

2018

Jeff Davis writes postgres-extension.rs as a conference POC

2019

Eric Ridge unveils PGX

2020

ZomboDB

2021

PGX transferred to TCDI

PGRX donated to Postgres Foundation

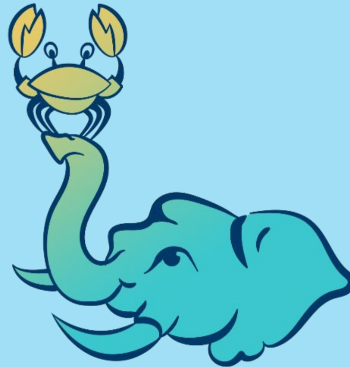
2023

aws

DALIBO

You got involved here?

What does PGRX get you?



A fully managed development environment ...

- cargo pgrx new: Create new extensions quickly
- cargo pgrx init: Install new (or register existing) PostgreSQL installs
- cargo pgrx run: Run your extension and interactively test it in psql (or pgcli)
- cargo pgrx test: Unit-test your extension across multiple PostgreSQL versions
- cargo pgrx package: Create installation packages for your extension

Target Multiple Postgres Versions

- Support from Postgres 12 to Postgres 17 from the same codebase
- Use Rust feature gating to use version-specific APIs
- Seamlessly test against all versions

Automatic Schema Generation

- Implement extensions entirely in Rust
- Automatic mapping for many Rust types into PostgreSQL
- SQL schemas generated automatically (or manually via `cargo pgrx schema`)
- Include custom SQL with `extension_sql!` & `extension_sql_file!`

Safety First

- Translates Rust `panic!` into Postgres `ERROR` that abort the transaction, not the process
- Memory Management follows Rust's drop semantics, even in the face of `panic!` and `elog(ERROR)`
- `#[pg_guard]` procedural macro to ensure the above
- Postgres Datums are `Option<T> where T: FromDatum`
- NULL Datums are safely represented as `Option::<T>::None`

First-class UDF support

- Annotate functions with #[pg_extern] to expose them to Postgres
- Return pgrx::iter::SetOfIterator<'a, T>for RETURNS SETOF
- Return pgrx::iter::TableIterator<'a, T>for RETURNS TABLE (...)
- Create trigger functions with #[pg_trigger]

Easy Custom Types

- `#[derive(PostgresType)]` to use a Rust struct as a Postgres type
  - represented as a CBOR-encoded object in-memory/on-disk
  - Represented as JSON when it needs to be human-readable
- `#[derive(PostgresEnum)]` to use a Rust enum as a Postgres enum
- Composite types supported with the `pgrx::composite_type!()` macro

Advanced Features

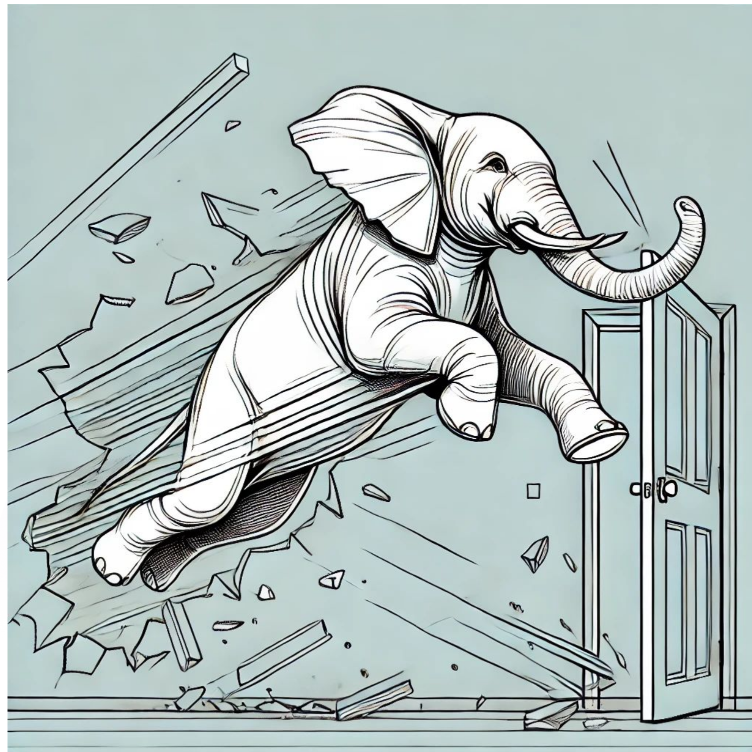
- Safe access to Server Programming Interface (SPI)
- Safe access to MemoryContext system via `pgrx::PgMemoryContexts`
- Executor/planner/transaction/subtransaction hooks
- Safely use Postgres-provided pointers with `pgrx::PgBox<T>`
- Access Postgres' logging system through macros
- Direct unsafe access to large parts of Postgres internals via the `pgrx::pg_sys` module
- Implement background workers, wal decoders, use shared memory ...

```rust
use pgrx::prelude::*;

::pgrx::pg_module_magic!();

#[pg_extern]
fn hello_world() -> &'static str {
    "Hello, PGConf.eu!"
}
```

Why not just use **C**?

Why Not use C indeed …

- ✅ No segfaults.
- ✅ No buffer overflows.
- ✅ No null pointers.
- ✅ No data races.
- ✅ Powerful type system.
- ✅ Unified build system.
- ✅ Dependency management.
- ✅ Wonderful extension ergonomics.



My PGRX Journey

- Architected, deployed and supported Postgres solutions
- Wrote (hacky) code in Python
- Wanted to write extensions, but hadn't used C professionally
- Found Rust (❤️)
- Able to write extensions

Timescale’s PGRX Journey

- Built TimescaleDB in C
- Wrote `timescaledb_toolkit` + `promscale` extensions in Rust
- Continued to add TimescaleDB features in C ...
- Wrote `pgvectorscale` with Rust
- Continued to add TimescaleDB features in C ...

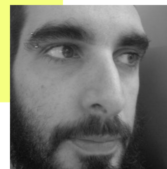
I moved the PostgreSQL Anonymizer extension to PGRX recently and rewrote about 1000 lines of C code in a few weeks of work without any prior knowledge of Rust.

The immediate benefits were :

- development comfort
- improved unit testing
- use of high-level Rust libraries (faker-rs)
- stability ("goodbye segfaults!")

Overall developing a Postgres extension is a huge responsibility because users might run your code on their production instances.

Thanks to the Rust safety model, they have the guarantee that a bug within the extension will not crash the entire instance.



Damien Clochard,
co-founder Dalibo

pgrx has been a joy to work with, happy to see it advertised to the broader community.

My application involves some custom types and a planner hook.

It took me less than 2 months to convert ~1000 lines of C extension to rust, add a ton of new features, and fix several insidious bugs along the way.

I wound up with ~4000 lines of rust including ~2000 lines of tests, and way more confidence that it does the right thing.

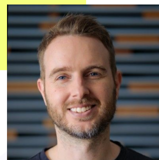
Will Murnane

We wouldn't have been able to release pgvectorscale as quickly or fearlessly without PGRX and Rust. The extension development tooling and Rust stdlib and crate ecosystem are second to none.

We were able to build pgvectorscale 5x faster which is the difference between having a product fully released in a quarter and year!

Matvey Arye,
Founding Engineer
Timescale

PGRX is Awesome



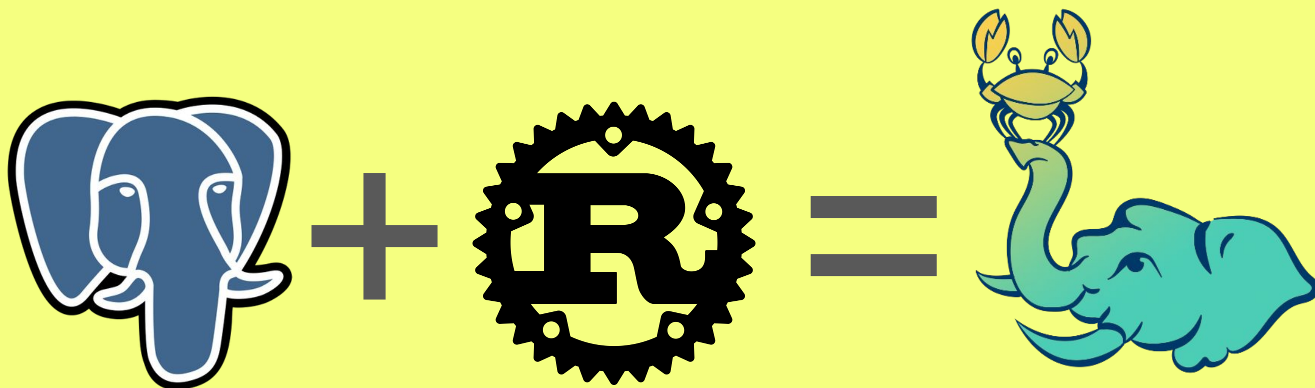
Paul Copplestone
CEO, Supabase

PGRX took my extension to the next level when I migrated the C to Rust.

Also I could sleep at night, not waiting for a hidden C-based segfault I never thought about taking out production

Matth

A Demo speaks 1000 words



In the live demo the benchmark was slow -> I forgot to run with –release mode!

perhaps I forgot to read the room

PG changes could also help PGRX

- Threading in Postgres is 🐉
- A major improvement would be

`Sigprocmask -> pthread_sigmask`

The use of **sigprocmask**() is unspecified in a multithreaded process; see pthread_sigmask(3).

Tom Lane said:
> I think the short answer about threading in bgworkers (or any other
> backend process) is "we don't support it; if you try it and it breaks,
> which it likely will, you get to keep both pieces".

Postgres Needs You
(maybe not you, other yous)

PG curious

Learn via C

Learn via Rust

PG contrib

If you’re interested, come play!

Github: https://github.com/pgcentralfoundation/pgrx
Github examples: ./pgrx_examples
Discord: https://discord.gg/hPb93Y9
Twitter: @pgrx_rs

Thank-you (and happy PGRXing)

Snappity snap,
we would love your feedback!