MARCH 18, 2024 BY PINARAF

Look ma, I wrote a new JIT compiler for PostgreSQL

Sometimes, I don't know why I do things. It's one of these times. A few months ago, Python 3.13 got its JIT engine, built with a new JIT compiler construction methodology (copy-patch, cf. research paper). After reading the paper, I was sold and I just had to try it with PostgreSQL. And what a fun ride it's been so far. This blog post will not cover everything, and I prefer other communication methods, but I would like to introduce pg-copyjit, the latest and shiniest way to ~~destroy and segfault~~ speed up your PostgreSQL server.

Before going any further, a mandatory warning: all code produced here is experimental. Please. I want to hear reports from you, like "ho it's fun", "ho I got this performance boost", "hey maybe this could be done", but not "hey, your extension cost me hours of downtime on my business critical application". Anyway, its current state is for professional hackers, I hope you know better than trusting experimental code with a production server.

In the beginning, there was no JIT, and then came the LLVM JIT compiler

In a PostgreSQL release a long time ago, in a galaxy far far away, Andres Freund introduced the PostgreSQL world to the magics of JIT compilation, using LLVM. They married and there was much rejoicing. Alas, darkness there was in the bright castle, for LLVM is a very very demanding husband.

LLVM is a great compilation framework. Its optimizer produces very good and efficient code, and Andres went further than what anybody else would have thought and tried in order to squeeze the last microsecond of performance in his JIT compiler. This is a wonderful work and I don't know how to express my love for the madness this kind of dedication to performance is. But LLVM has a big downside : it's not built for JIT compilation. At least not in the way PostgreSQL will use it: the LLVM optimizer is very expensive, but not using it may be worse than no compilation at all. And in order to compile only the good stuff, the queries that can enjoy the performance boost, the typical query cost estimation is used. And that's the PostgreSQL downside making the whole thing almost impossible: costs in PostgreSQL are not designed to mean anything. They are meant to be compared to each other, but do not mean anything regarding the real execution time. A query costing 100 may run in 1 second, while another costing 1000 may run in 100 milliseconds. It's not a bug, it's a design decision. That's why a lot of people (including me) end up turning off the JIT compiler: most if not all queries on my production system will not get enough from the performance boost to compensate the LLVM optimizer cost. If I can run the query 10ms faster but it needed 50ms to be optimized, it's pure loss.

There is one way to make the LLVM JIT compiler more usable, but I fear it's going to take years to be implemented: being able to cache and reuse compiled queries. I will not dig further into that topic in this post, but trust me, it's not going to be a small feat to achieve.

And in 2021, copy-and-patch was described…

So, what can we do? We need fast enough code generated the fastest way possible. Fast enough code mean at least a bit faster than the current interpreter… But writing a compiler is painful, writing several code generators (for different ISAs for instance) is even worse…

This is where the innovation of copy-and-patch comes into play and saves the day.

With copy-patch, you write stencils in C. These stencils are functions with holes, and they are compiled by your typical clang compiler (gcc support pending, too complicated to explain here). Then when you want to compile something, you stitch stencils together, fill in the gaps, and jump straight into your brand new "compiled" function.

And this is it. This is the magic of copy-and-patch. You only copy the stencils in a new memory area, patch the holes, and voilà.

Of course, you can go further. You can figure out what computation can be done at compilation time, you can split loops in several stencils to unroll them, you can merge several stencils together to optimize them in one go (creating kind of meta-stencils…)

This paper caught the eyes of the Faster-CPython team, they implemented it in CPython 3.13, and this is when more people (including me) discovered it.

Bringing copy-and-patch to PostgreSQL

So, what does it take to build a new JIT engine in PostgreSQL? Hopefully, not that much, otherwise I would likely not be blogging about this.

When JIT compilation was introduced, it was suggested on hackers to make LLVM a plugin, allowing future extensions to bring other JIT compilers. Back then, I was quite skeptical to this idea (but never expressed this opinion, I did not want to be wrong later), and it turned out I proved myself wrong… The interface is really simple, your .so only needs to provide a single _PG_jit_provider_init function, and in this function initialize three callbacks, named compile_expr, release_context and reset_after_error. The main one is obviously compile_expr. You get one ExprState* parameter, a pointer to an expression, made of opcodes. Then it's "only" a matter of compiling the opcodes together in any way you want, mark this built code as executable, and changing the evalfunc to this code instead of the PostgreSQL interpreter. This is easy, and you have an automatic fallback to the PostgreSQL interpreter if you encounter any opcode you've not implemented yet.

The copy and patch algorithm (implemented with only a few small optimizations so far) is so easy I can explain it here. For each opcode, the compiler will look into the stencil collection. If the opcode has a stencil, the stencil is appended to the "built" code. Otherwise, the compilation stops and the PostgreSQL interpreter will kick in. After appending the stencil, each of its holes are patched with the required value.

For instance, let's consider this basic unoptimized stencil, for the opcode CONST.

```
Datum stencil_EEOP_CONST (struct ExprState *expression, struct ExprContext
*econtext, bool *isNull)
{
    *op.resnull = op.d.constval.isnull;
    *op.resvalue = op.d.constval.value;

    NEXT_OP();
}
```

op is declared as extern ExprEvalStep op; (and NEXT_OP is a bit harder to explain, I won't dig into it here). When building this to a single .o file, the compiler will leave a hole in the assembly code, where the address for op will have to be inserted (using a relocation). When the stencil collection is built, this information is kept and used by the JIT compiler to use the current opcode structure address in order to get a working code.

The build process for the stencils is quite fun, not complicated, but fun. The first step is to build the stencils to a single .o file, and then extract the assembly code and relocations from this .o file into C usable structures, that the JIT compiler will link to.

And that's about all there is.

At first, I was extracting the assembly code manually. Using that way, I managed to get the three needed opcodes for SELECT 42; to work. And there was much joy. After this first proof of concept (and I guess some disturbed looks a few days ago at PgDay.Paris when people saw me happy with being able to run SELECT 42, that may have sound weird), I wrote a DirtyPython (unofficial variant) script to automate the assembly code extraction, and in a few hours I implemented function calls, single table queries, more complicated data types, introduced a few optimizations…

Current state

It works on my computer with PostgreSQL 16. It should be fine with older releases. It only supports AMD64 because that's what I have and I can not target everything at once. Later I will add ARM64, and I would love to have some time to add support for some interesting targets like POWER64 or S390x (these may require some compiler patches, sadly, and access to such computers, nudge nudge wink wink)…

Performance-wise, well, keeping in mind that I've spent almost no time optimizing it, the results are great. Code generation is done in a few hundreds microseconds, making it usable even for short queries, where LLVM is simply out of the game. On a simple SELECT 42; query, running with no JIT takes 0,3ms, with copyjit it requires 0,6ms, LLVM with no optimizations goes to 1,6ms and optimizing LLVM will require 6,6ms. Sure, LLVM can create really fast code, but the whole idea here is to quickly generate fast enough code, and thus comparing both tools won't make much sense.

But still, you are all waiting for a benchmark, so here we go, benchmarking two queries on a simple non-indexed 90k rows table. This benchmark is done on a laptop and my trust in such a benchmark setup is moderate at best, a proper benchmark will be done later on a desktop computer without any kind of thermal envelope shenanigans. And I have not optimized my compiler, it's still quite stupid and there is a lot of things that can and must be done.

Query	Min/max (ms)	Median (ms) and stdev
select * from b; — no JIT	10.340/14.046	10.652/0.515

select * from b; — JIT	10.326/14.613	10.614/0.780
select i, j from b where i < 10; — no JIT	3.348/4.070	3.7333/0.073
select i, j from b where i < 10; — JIT	3.210/4.701	3.519/0.107

Stupid benchmark on a laptop running non-optimized code, don't trust these…

As you can see, even in the current unfinished state, as soon as there is CPU work to do (here it's the where clause), performance relative to the interpreter get better. It's only logical, and what is important here is that even if the JIT is an extra, slightly time consuming step, it takes so little time even these queries can go a few percents faster.

Note that even if I've implemented only a small handful of opcodes, I can run any query on my server, the JIT engine will only complain loudly about it and let the interpreter run the query…

For the more curious, the code is dumped there on github. I said dumped because I focus only on the code and not on the clarity of my git history nor on wrapping it in a nice paper with flying colors and pretty flowers, that's what you do when the code is done, this one isn't yet… If you want to build it, the build-stencils.sh file must be run manually first. But again, I do not document it yet because I simply can not provide any support for the code in its current state.

TODO…

This is a proof of concept. I've not worked on making it easy to build, on making it possible to package it… The build scripts are Debian and PostgreSQL 16 specific. And, well, to be honest, at this point, I don't care much and it will not trouble me, my focus is on implementing more opcodes, and searching for optimizations.

I really hope I will reach a point where I can safely package this and deploy it on my production servers. This way, I'll keep using the LLVM JIT on the server that can use it (a GIS server where queries are worth the optimization) and use this JIT on my web-application databases, where short query time is a must have, and the LLVM optimizations end up being counter-productive.

I am also dead serious on porting this to other architectures. I love the old days of Alpha, Itanium, Sparc, M68k and other different architectures. I am not going to use this kind of system, but I miss the diversity, and I really don't want to be a part of the monoculture issue here.

Thanks

First, huge thanks to my current day-job employer, Entr'ouvert. We are a small french SaaS company, free-software focused, and my colleagues simply let me toy on this between tickets and other DBA or sysadmin tasks.

I would like to thank my DBA friends for supporting me and motivating me into doing this (won't give their names, they know who they are). BTW: use PoWA, great tool, tell your friends…

Also, quick question: they suggest I shall go to PGConf.dev to show this, but it's too late for the schedule and since I live in France I did not intend to go there. If you think it's important or worth it, please, please, say so (comments below, or my email is p@this.domain), otherwise see you in future european PG events 🙂

POSTGRESQL

JIT, POSTGRESQL

3 Replies to "Look ma, I wrote a new JIT compiler for PostgreSQL"

Xing Guo

MARCH 18, 2024 AT 1:06 PM

Nice post! I learned a lot from your codes. It's definitely worth sharing it on pgconf!

P.S. I've also tried to implement JIT provider prototypes for PostgreSQL for fun!

– https://github.com/higuoxing/pg_slowjit
– https://github.com/higuoxing/pg_asmjit

Pingback: Look ma, I wrote a new JIT compiler for PostgreSQL – Cyber Geeks Global

Pingback: [illegible]PostgreSQL[illegible] - [illegible]