JIT compilation in PostgreSQL

From jit=off to jit_above_cost=0

Current state

- LLVM-based JIT in core since PG11
- Used to compile expressions
- Can also compile ‘tuple deforming’
- Triggered based on costs
- Built with OLAP in mind

Current state

- Issues :
  - LLVM is one huge beast
  - Its optimizer is built for AOT, not JIT
  - -O0 can produce terrible code (slower than interpreting)
  - Cost and runtime don’t have a good correlation => JIT triggered for queries with no possible gains

New player in town : copyjit

- Based on ‘copy-and-patch’ paper
- Objectives :
  - very fast compilation time
  - good enough code
  - Web applications, OLTP, not OLAP

How it works

- Ahead of time, a collection of stencils is built
- Each stencil is a piece of code with holes
- When a query must be compiled, stencils are put together in memory
- And holes are filled in

How it works

```c
extern ExprEvalStep op;
extern void CONST_ISNULL;
extern intptr_t CONST_VALUE;

Datum stencil_EEOP_CONST (struct ExprState *state, struct ExprContext *econtext, bool *isNull)
{
    *op.resnull = (char) ((intptr_t) &CONST_ISNULL);
    *op.resvalue = (Datum) &CONST_VALUE;

    goto_next;
}

Datum stencil_EEOP_DONE (struct ExprState *state, struct ExprContext *econtext, bool *isNull)
{
    *isNull = state->resnull;
    return state->resvalue;
}
```

Stencils required for ‘SELECT 42’

```
0000000000000000 <stencil_EEOP_DONE>:
 movzbl 0x5(%rdi),%eax
 mov    %al,(%rdx)
 mov    0x8(%rdi),%rax
 ret

0000000000000010 <stencil_EEOP_CONST>:
 movabs $0x0,%rax
 mov    0x10(%rax),%rcx
 movabs $0x0,%r8
 test   %r8b,%r8b
 setne  (%rcx)
 mov    0x8(%rax),%rax
 movabs $0x0,%rcx
 mov    %rcx,(%rax)
 movabs $0x0,%rax
 jmp    *%rax
```

Corresponding amd64 assembly

How it works

```
0000000000000000 <stencil_EEOP_DONE>:
 movzbl 0x5(%rdi),%eax
 mov    %al,(%rdx)
 mov    0x8(%rdi),%rax
 ret

0000000000000010 <stencil_EEOP_CONST>:
 movabs $0x0,%rax
 mov    0x10(%rax),%rcx
 movabs $0x0,%r8
 test   %r8b,%r8b
 setne  (%rcx)
 mov    0x8(%rax),%rax
 movabs $0x0,%rcx
 mov    %rcx,(%rax)
 movabs $0x0,%rax
 jmp    *%rax
```

Corresponding amd64 assembly

```
const unsigned char EEOP_DONE__code[11] = {0xf, 0xb6, 0x47,
0x8, 0xc3};
// No patch for EEOP_DONE
const unsigned char EEOP_CONST__code[47] = {0x48, 0xb8, 0x0
0x0, 0x48, 0x8b, 0x48, 0x10, 0x49, 0xb8, 0x0, 0x0, 0x0, 0x0
0xc0, 0xf, 0x95, 0x1, 0x48, 0x8b, 0x40, 0x8, 0x48, 0xb9, 0x
0x0, 0x48, 0x89, 0x8};
const Patch EEOP_CONST__patches[3] = {
{2, RELKIND_R_X86_64_64, TARGET_OP},
{16, RELKIND_R_X86_64_64, TARGET_CONST_ISNULL},
{36, RELKIND_R_X86_64_64, TARGET_CONST_VALUE},
};
```

C structures for the compiler

Compilation time

- Under 100 us even for more complicated queries
- Not even optimized yet...

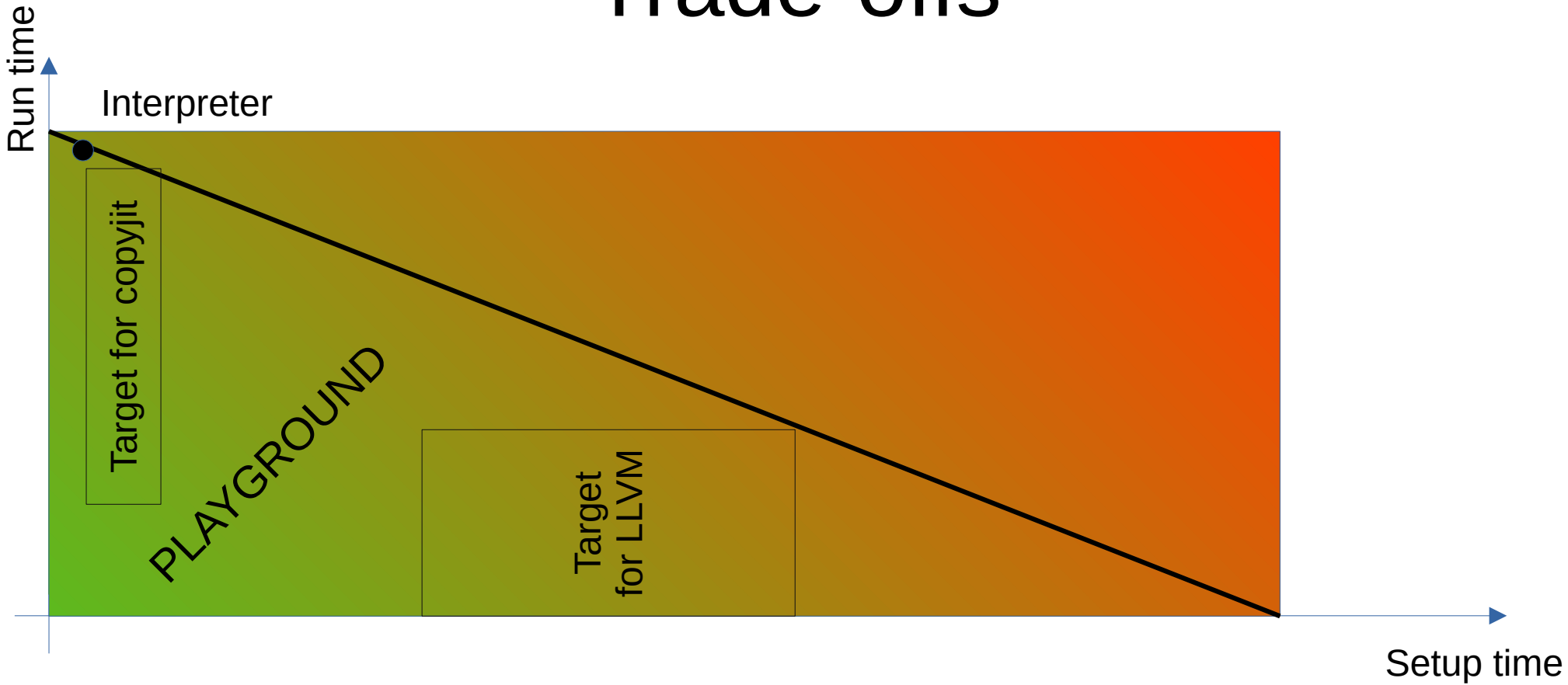
Optimizing the generated code

- Three options :
  - Create specialized stencils
  - Create micro-stencils
  - Create stencils spanning several ops
- Examples :
  - Calling int4eq is common, create a specific stencil
  - When calling a strict function, unroll the null checks

Work in progress

- Working on tuple deforming
  - Without this, ‘only’ 5 to 10 % gain in query execution
- All opcodes are not implemented yet
- AMD64 specific so far
  - Ready for ARM64, for others see below...
- Depends on Clang/LLVM and musttail
  - Thus limited architecture support there

Trade-offs


Run time
Interpreter
Target for copyjit
PLAYGROUND
Target
for LLVM
Setup time

Open for competition !

- I’m certain some other solutions could be tested too
  - Cranelift ? ‘Cranelift is a fast, secure, relatively simple and innovative compiler backend.’
- Could/should we have PostgreSQL support tiered compilation ?
  - Depending on query cost or estimated rows, call different JIT compilers…
- What is needed on PostgreSQL side to cache compiled code ?