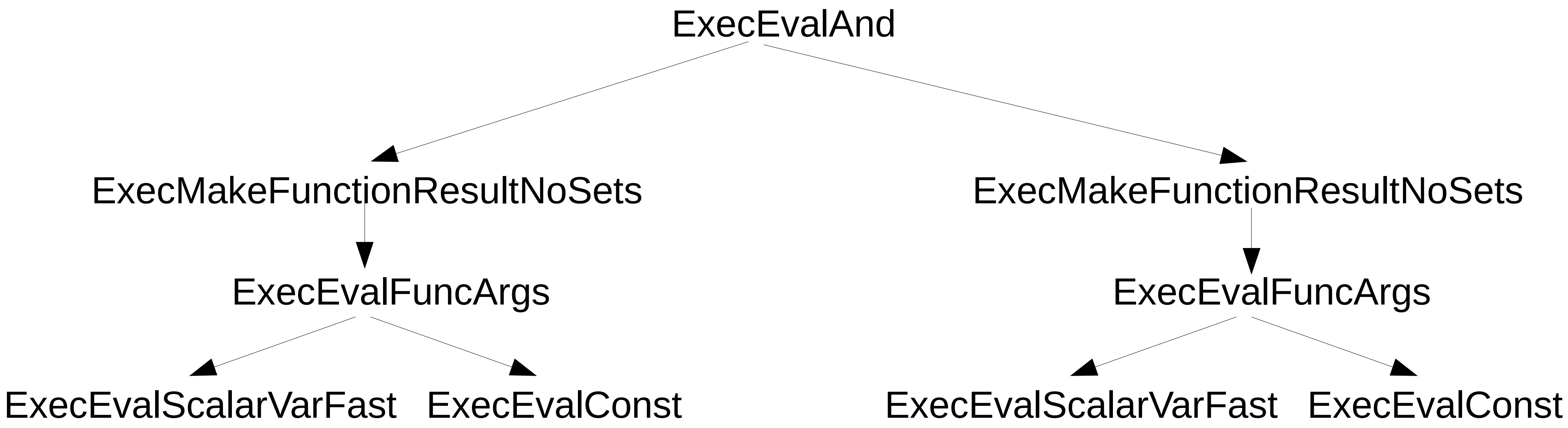
Integrating Just In Time Compilation

Andres Freund
PostgreSQL Developer & Committer
andres@anarazel.de andres@citusdata.com
Citus Data – citusdata.com - @citusdata

http://anarazel.de/talks/pgcon-2017-05-25/jit-pgcon-2017-05-25.pdf

Motivation



ExecEvalAnd

ExecMakeFunctionResultNoSets

ExecEvalFuncArgs

ExecEvalScalarVarFast

ExecEvalConst

ExecMakeFunctionResultNoSets

ExecEvalFuncArgs

ExecEvalScalarVarFast

ExecEvalConst

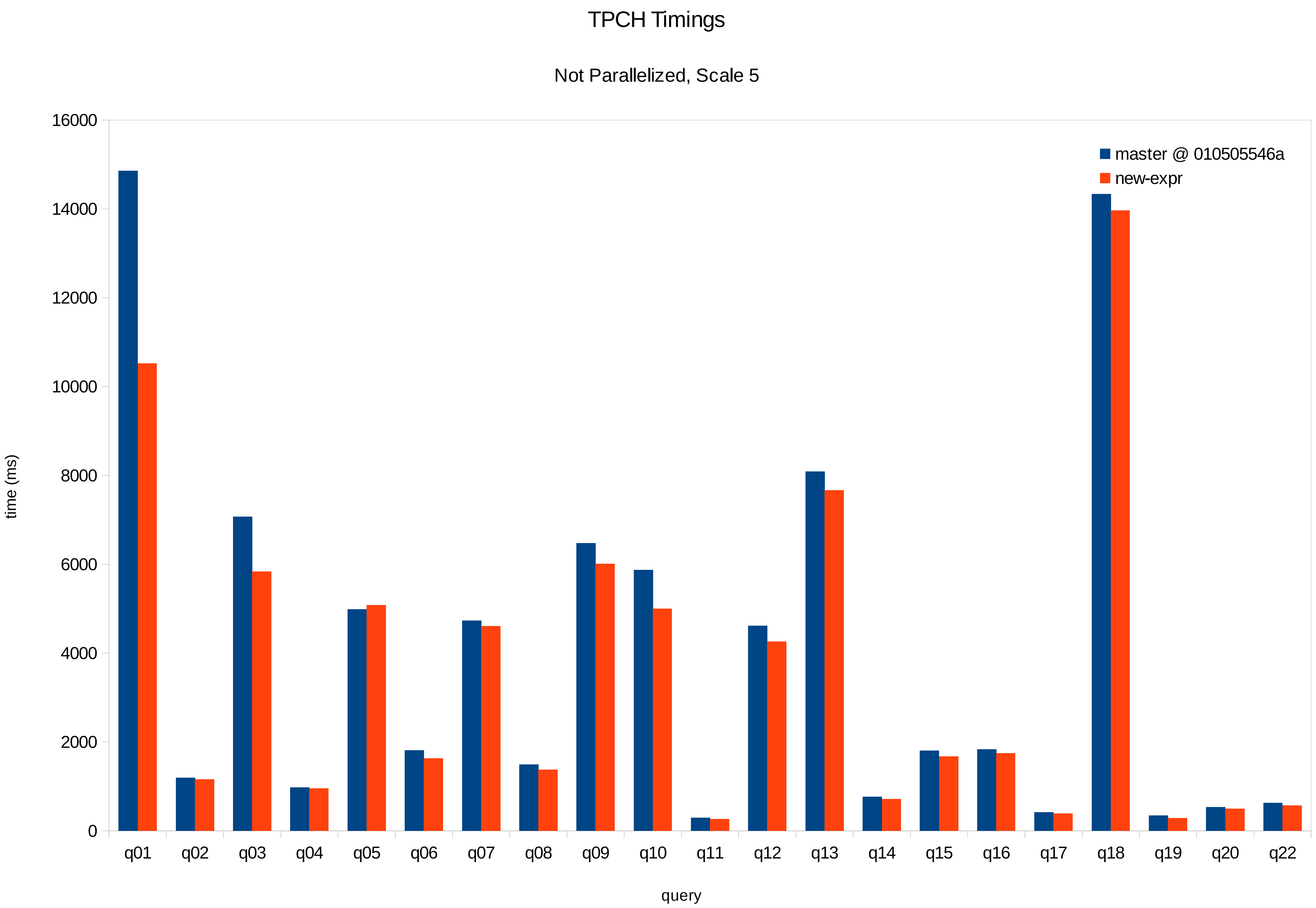
>= v10 Expression Evaluation

```
static Datum
ExecInterpExpr(ExprState *state, ExprContext *econtext, bool *isnull)
{
...
    while (true)
    {
        switch (op→opcode)
...
            case EEOP_CONST:
            {
                *op->resnull = op->d.constval.isnull;
                *op->resvalue = op→d.constval.value;

                op++;
                continue;
            }
```

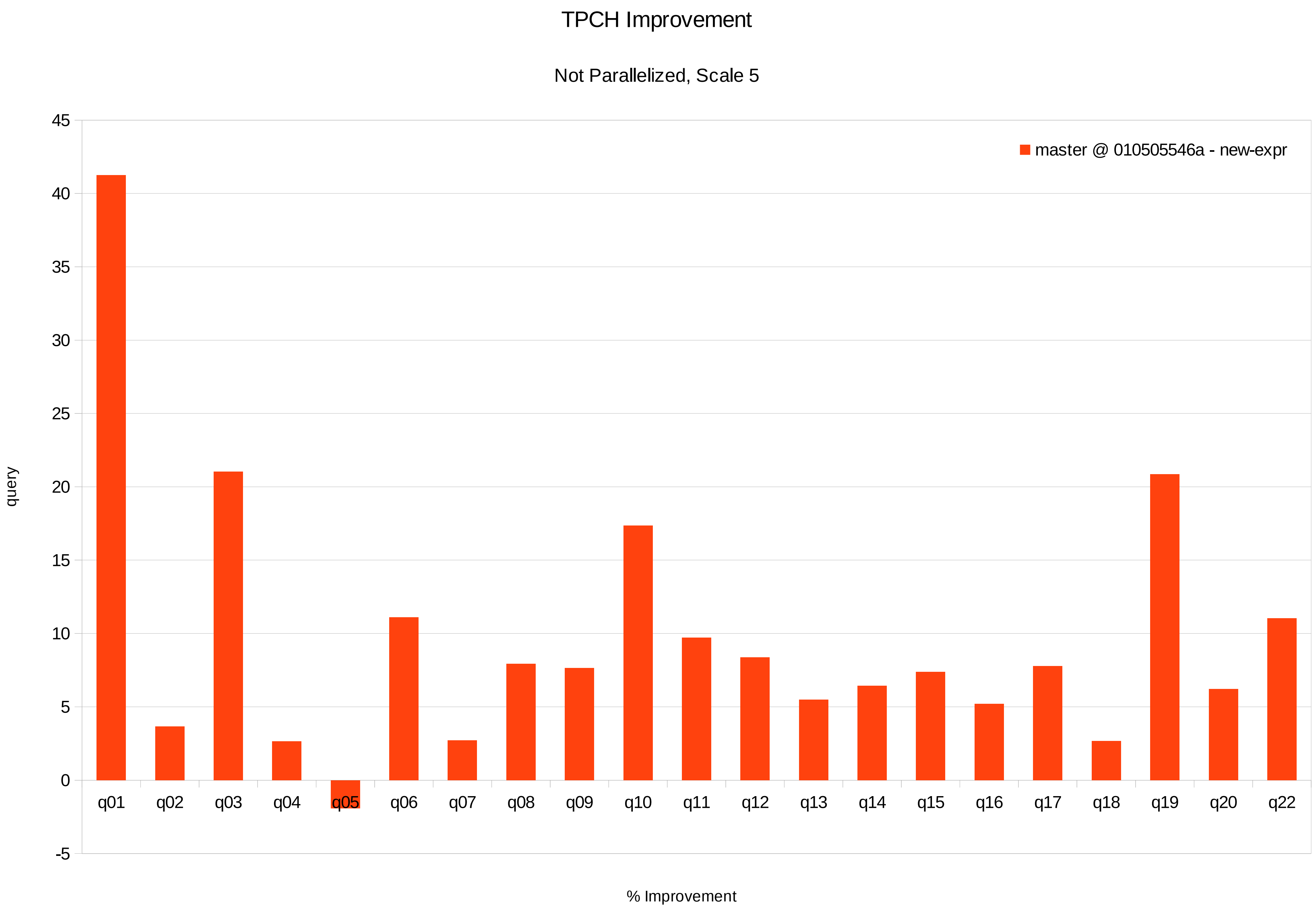
>= v10 Expression Evaluation

```
case EEOP_FUNCEXPR:
{
    FunctionCallInfo fcinfo = op->d.func.fcinfo_data;

    fcinfo->isnull = false;
    *op->resvalue = (op->d.func.fn_addr) (fcinfo);
    *op->resnull = fcinfo->isnull;

    op++;
    continue;
}
```

```
case EEOP_BOOL_AND_STEP_LAST:
{
    if (*op->resnull)
    {
        /* result is already set to NULL, need not change it */
    }
    else if (!DatumGetBool(*op->resvalue))
    {
        /* result is already set to FALSE, need not change it */

        /*
         * No point jumping early to jumpdone - would be same target
         * (as this is the last argument to the AND expression),
         * except more expensive.
         */
    }
    else if (*op->d.boolexpr.anynull)
    {
        *op->resvalue = (Datum) 0;
        *op->resnull = true;
    }
    else
    {
        /* result is already set to TRUE, need not change it */
    }

    EEO_NEXT();
}
```


TPCH Timings
Not Parallelized, Scale 5
time (ms)
16000
14000
12000
10000
8000
6000
4000
2000
0
q01
q02
q03
q04
q05
q06
q07
q08
q09
q10
q11
q12
q13
q14
q15
q16
q17
q18
q19
q20
q22
query
master @ 010505546a
new-expr

citusdata


TPCH Improvement
Not Parallelized, Scale 5
45
40
35
30
25
20
15
10
5
0
-5
query
master @ 010505546a - new-expr
q01
q02
q03
q04
q05
q06
q07
q08
q09
q10
q11
q12
q13
q14
q15
q16
q17
q18
q19
q20
q22
% Improvement

citusdata

>= v10 Expression Evaluation

- a lot faster in v10
  - More importantly preparation for JITing
- still massive bottleneck for many use-cases
  - for many rows
  - instantiation overhead (fix not talked about here)
- minor details can be micro-optimized further, but no large further improvements
- biggest problem: jumps, branches, function calls

What’s JIT

- Interpreted Code → Native Code
- Ahead of time: gcc, clang, msvc
- Just in Time:
  - Java
  - Javascript in common browsers
  - Luajit, …
- Process:
  - Generate native code
  - Load as Executable Memory
  - Execute (Function Pointer)

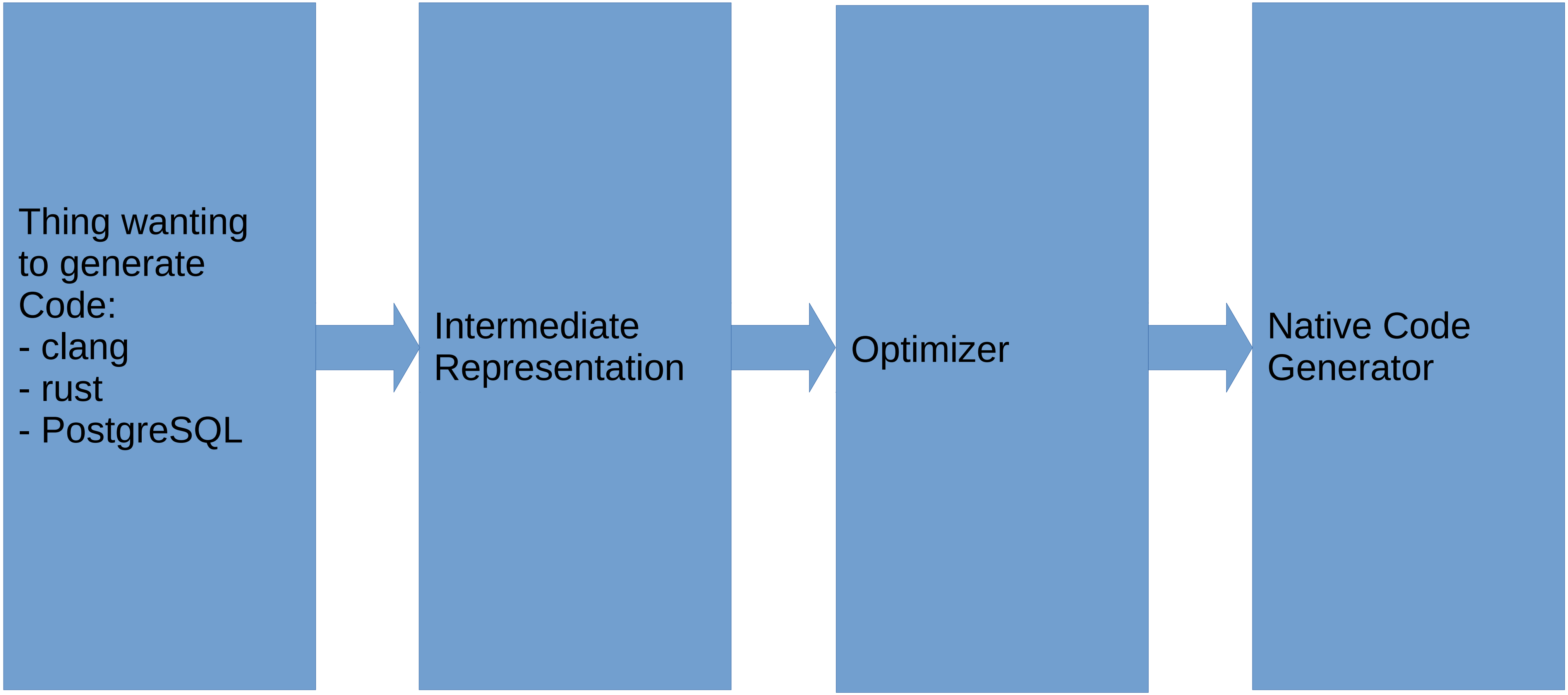
Why JIT

- Interpretation has a lot of “jumps”
- Interpretation calls a lot of “unknown functions”
- Native Code doesn’t have those Issues
- Only do so when beneficial
  - Generating a native function is expensive (~0.1-1ms + optimization)
- Can be used in a lot of places

LLVM

- llvm.org
- Formerly known as: low-level-virtual-machine
- Compiler Framework
- Used by:
  - Clang
  - Swift
  - Rust
  - …
- Intermediate Language/Representation

LLVM “Flow”



Thing wanting to generate Code:
- clang
- rust
- PostgreSQL

Intermediate Representation

Optimizer

Native Code Generator

LLVM IR

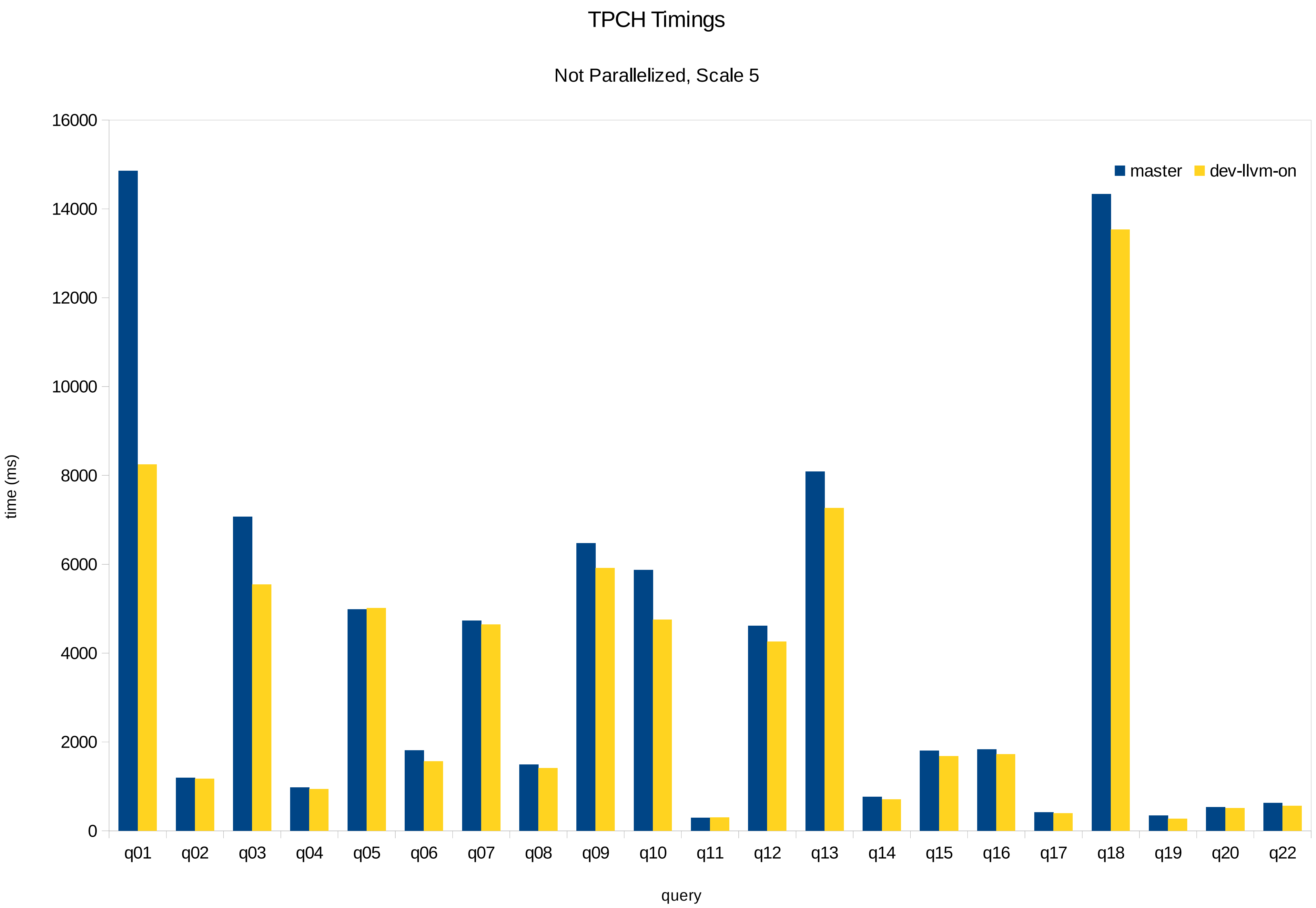
```
%struct.FunctionCallInfoData = type { %struct.FmgrInfo*, %struct.Node*, %struct.Node*, i32, i8, i16, [100 x
i64], [100 x i8] }

%struct.ExprContext = type { i32, %TupleTableSlot*, %TupleTableSlot*, %TupleTableSlot*, i64*, i64*, i64*,
i64*, i64*, i8*, i64, i8, i64, i8, i64*, i64* }

define i64 @evalexpr4(%struct.ExprState*, %struct.ExprContext*, i8*) {
entry:
  %v.state.resvalue = getelementptr inbounds %struct.ExprState, %struct.ExprState* %0, i32 0, i32 3
  %v.state.resnull = getelementptr inbounds %struct.ExprState, %struct.ExprState* %0, i32 0, i32 2
  %3 = getelementptr inbounds %struct.ExprContext, %struct.ExprContext* %1, i32 0, i32 1
..
  br label %block.op.0.start

block.op.0.start:                                 ; preds = %entry
  %17 = call i64 @slot_getsomeattrs(%TupleTableSlot* %v_innerslot, i32 1)
  br label %block.op.1.start
…
block.op.25.start:                                ; preds = %block.op.24.qualfail, %block.op.24.start
  %128 = load i64, i64* %v.state.resvalue
  %129 = load i8, i8* %v.state.resnull
  store i8 %129, i8* %2
  ret i64 %128
}
```

LLVM IR Generation

```
case EEOP_CONST:
    {
        LLVMValueRef v_constvalue, v_constnull;

        v_constvalue = LLVMConstInt(TypeSizeT,
                op->d.constval.value, false);
        v_constnull = LLVMConstInt(LLVMInt8Type(),
                op->d.constval.isnull, false);

        LLVMBuildStore(builder, v_constvalue, v_resvaluep);
        LLVMBuildStore(builder, v_constnull, v_resnullp);

        LLVMBuildBr(builder, opblocks[i + 1]);
        break;
    }
```

LLVM IR Generation

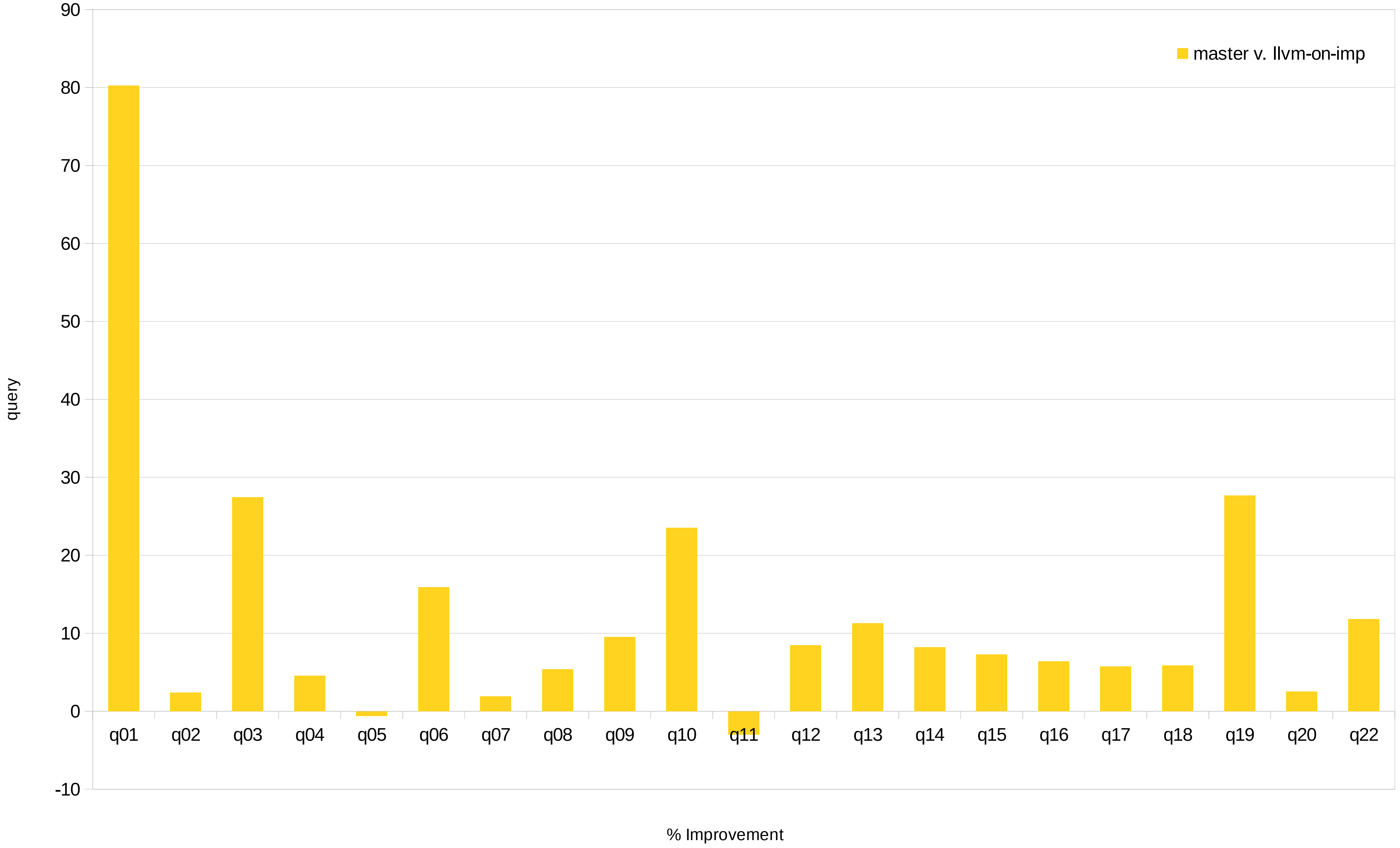
```
    case EEOP_WHOLEROW:
        {
…
           v_params[0] = v_state;
           v_params[1] = LLVMBuildIntToPtr(builder,
                  LLVMConstInt(TypeExprEvalOp, (uintptr_t) op, false),
                  LLVMPointerType(TypeSizeT, 0),
                  "");
           v_params[2] = v_econtext;
           LLVMBuildCall(builder, l_EvalWholeRowVar,
                      v_params, lengthof(v_params), "");
           LLVMBuildBr(builder, opblocks[i + 1]);
           break;
        }
```



TPCH Timings
Not Parallelized, Scale 5
time (ms)
query
■ master
■ dev-llvm-on
0
2000
4000
6000
8000
10000
12000
14000
16000
q01
q02
q03
q04
q05
q06
q07
q08
q09
q10
q11
q12
q13
q14
q15
q16
q17
q18
q19
q20
q22

TPCH Improvement

Not Parallelized, Scale 5



citusdata

Issue: Planning

- JIT after jit_expressions_after = …
  - Slow for a while
  - JITs one-by-one
- JIT when jit_expressions_above_cost = ...
  - A bit weird
  - Costs not really comparable → hard to tune
  - JITs all functions at once
  - Not really JIT
- ?

Issue: syncing structs

```
members[ 0] = LLVMInt32Type(); /* tag */
snum_ExprState_tag = 0;
members[ 1] = LLVMInt8Type(); /* flags */
snum_ExprState_flags = 1;
members[ 2] = LLVMInt8Type(); /* resnull */
snum_ExprState_resnull = 2;
members[ 3] = TypeSizeT; /* resvalue */
snum_ExprState_resvalue = 3;

members[13] = LLVMPointerType(LLVMInt8Type(), 0);

StructExprState = LLVMStructCreateNamed(LLVMGetGlobalContext(),
                                        "struct.ExprState");
LLVMStructSetBody(StructExprState, members, lengthof(members), false);
```

Issue: LLVM API Stability

- Use C API, rather than C++, changes slower
- Fix limitations

Issue: JIT Overhead

- Prepared Statements
- Caching
- Nontrivial:
  - ExprState contains ephemeral pointers
  - Cache Management

Next Step: Inline Functions

- WHERE (a + b) < c
  - float8pl / int4pl / *pl
  - float8lt / int4lt / *lt
- AVG(a + b)
  - int8pl
  - int8_avg_accum
  - numeric_poly_avg
- Function calls are expensive, body often cheap to execute

Next Step: Inline Functions

- Generate Code: `clang -emit-llvm`
- Combine Functions: `llvm-link`
- Extract Useful: `llvm-lto -exported-symbol float8mul …`
- Use: Link/Merge Modules at JIT
- Increases JIT Compilation Time
- Speedup: TPCH up-to 2.2x, isolated: ~5x
- Extensions:
  - Ignore
  - Specify bitcode at CREATE FUNCTION
  - Embed in .so?

Issue: Not using ExecEvalExpr()

- Aggregates:
  - Transition Functions
    - Hotspot
    - Very poorly predicted
  - Final-Functions
- Hash-Join:
  - Each column evaluated separately (ExecEvalExpr)
  - Hash function “manually” invoked
- Hash-Agg/Grouping/Subplan/WITH RECURSIVE
  - projected “below”
  - Each column evaluated separately (slot_getattr)
  - Hash function “manually” invoked

Next Step: “Skipping” Deforming

- NOT NULL, fixed-with columns: constant offset
- Planner/Executor:
  - Need to maintain NOT NULLness
  - Need to maintain whether virtual tuple
- JIT
  - skip slot_getsomeattrs() if applicable
  - Replace w/ pointer magic
- Current State:
  - Works if not null, fixed width :)
  - TPCH: up to ~1.8x
- Next: JIT variable width deforming

Order of Sub-Tasks

- profiling support of JITed functions
  - Patches to LLVM submitted
  - How to name functions?
- LLVM infrastructure / resource management integration
- plain expression JITing
- function / operator function inlining
- direct access to deformed columns
- JITing deforming
- Memory-Lifetime hints & other codegen improvements

Integrating Just In Time Compilation

Andres Freund
PostgreSQL Developer & Committer
andres@anarazel.de andres@citusdata.com
Citus Data – citusdata.com - @citusdata

http://anarazel.de/talks/pgcon-2017-05-25/jit-pgcon-2017-05-25.pdf

citusdata