Speeding up query execution in PostgreSQL using LLVM JIT compiler

Dmitry Melnik
dm@ispras.ru

Institute for System Programming
of the Russian Academy of Sciences
(ISP RAS)

September 8, 2016

Speeding up PostgreSQL

- What exactly do we want to accelerate?
  - Complex queries where performance "bottleneck" is CPU rather than disk
    - OLAP, decision-making support, etc.
  - Goal: performance optimization on TPC-H benchmark
- How to achieve speedup?
  - Use LLVM MCJIT for just-in-time compilation of PostgreSQL queries

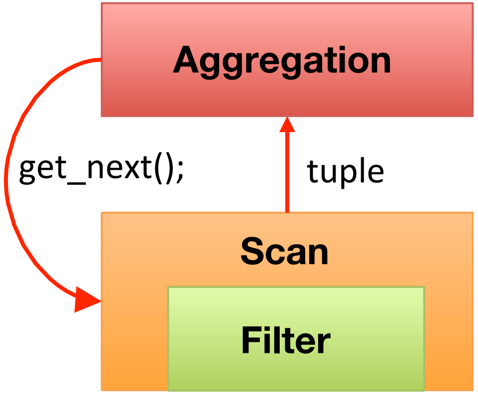
What if we add LLVM JIT to the PostgreSQL?

Example of query in Postgres

`SELECT COUNT(*) FROM tbl WHERE (x+y)>20;`

Aggregation — `COUNT(*)`
Scan — `FROM tbl`
Filter — `WHERE (x+y)>20;`

"Volcano-style" iterative model



get_next() - indirect call

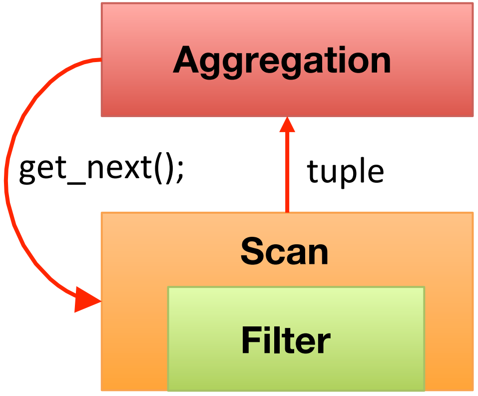
Example of query in Postgres

`SELECT COUNT(*) FROM tbl WHERE (x+y)>20;`

Aggregation **Scan** **Filter**

interpretation: 56% of execution time

"Volcano-style" iterative model



get_next() - indirect call

Example of query in Postgres

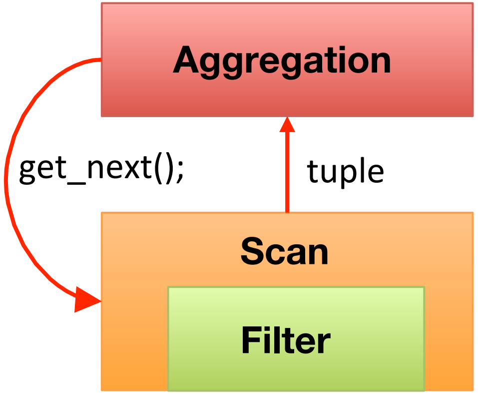
`SELECT` `COUNT(*)` `FROM tbl` `WHERE (x+y)>20;`

Aggregation **Scan** **Filter**

~~**interpretation:**~~
~~**56%** of execution~~

code, generated by LLVM:
6% of execution time

“Volcano-style” iterative model



get_next() - indirect call

Profiling TPC-H

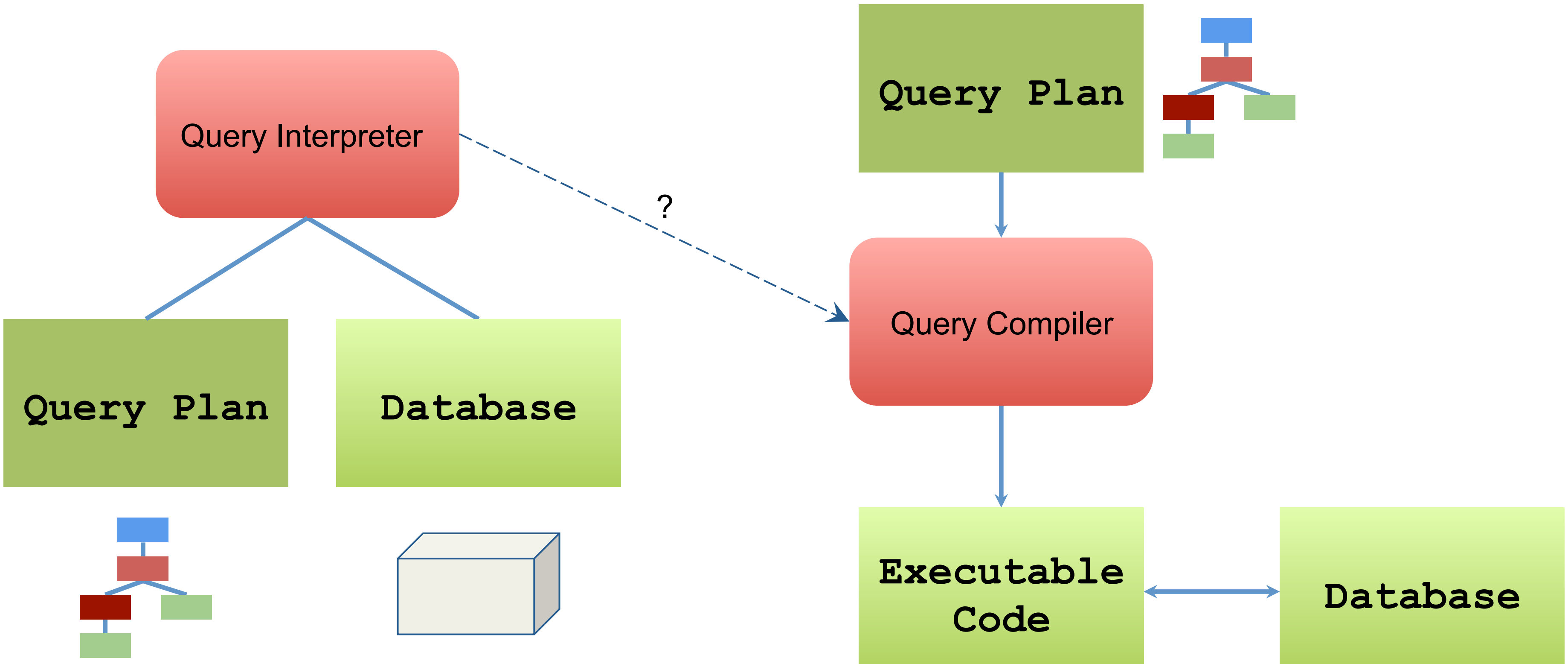
ISP RAS

TPC-H Q1:

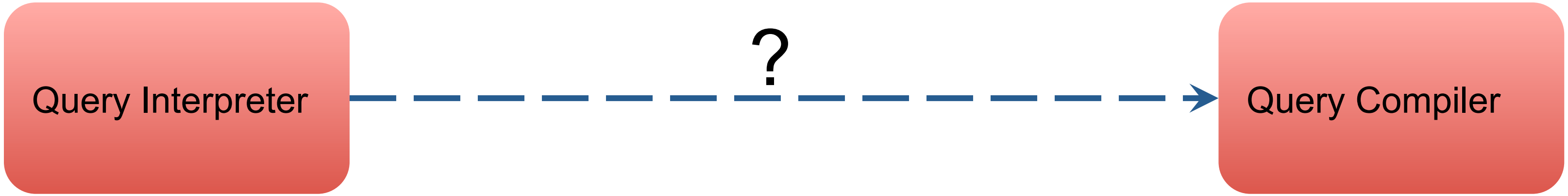
```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <=
        date '1998-12-01' -
        interval '60 days'
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

Function	TPC-H Q1	TPC-H Q2	TPC-H Q3	TPC-H Q6	TPC-H Q22	Average on TPC-H
ExecQual	6%	14%	32%	3%	72%	25%
ExecAgg	75%	-	1%	1%	2%	16%
SeqNext	6%	1%	33%	-	13%	17%
IndexNext	-	57%	-	-	19%	38%
BitmapHeapNext	-	-	-	85%	-	85%

Query Interpretation vs. Query Compilation (1)

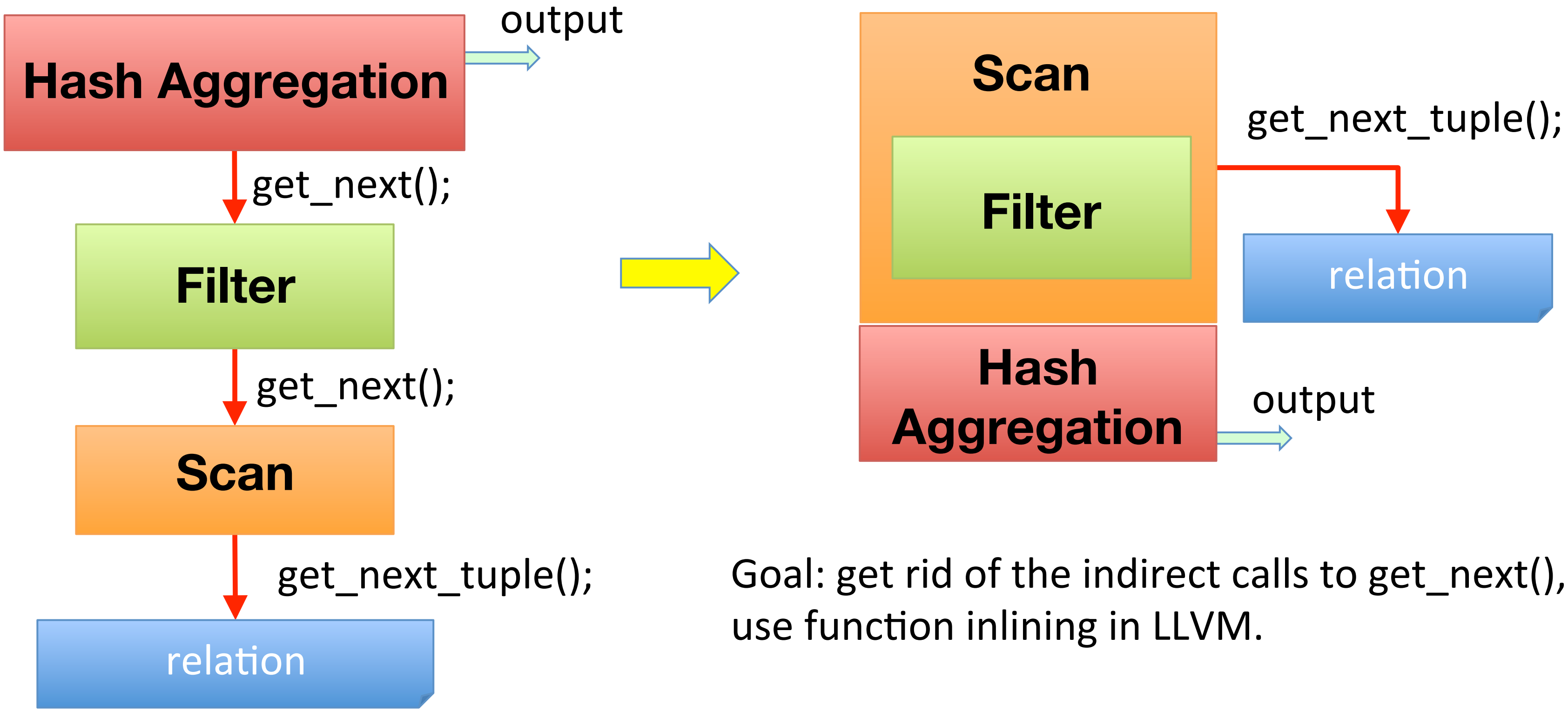


Query Interpretation vs. Query Compilation (2)



1. Reimplementing query interpreter is cumbersome and error-prone.
   a. It is necessary to implement code generation for all operations of all types supported in expressions (about 2000 functions in total).
2. Would need to constantly maintain and keep in sync.
3. <u>Ideally</u>: derive one from the other.

Getting rid of "Volcano-style" iterative model



Goal: get rid of the indirect calls to get_next(), use function inlining in LLVM.

Automatic code generation

int.c PostgreSQL backend file

```
Datum
int8pl(FunctionCallInfo fcinfo)
{
    int64       arg1 = fcinfo->arg[0];
    int64       arg2 = fcinfo->arg[1];
    int64       result;

    result = arg1 + arg2;

    /*
     * Overflow check.
     */
    if (SAMESIGN(arg1, arg2)
       && !SAMESIGN(result, arg1))
        ereport(ERROR,
       (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE),
                 errmsg("integer out of range")));
    PG_RETURN_INT64(result);
}
```

Clang

int.bc LLVM IR

```
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
entry:
  %1 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
  %4 = load i64, i64* %3
  %5 = add nsw i64 %4, %2
  %.lobit = lshr i64 %2, 63
  %.lobit1 = lshr i64 %4, 63
  %6 = icmp ne i64 %.lobit, %.lobit1
  %.lobit2 = lshr i64 %5, 31
  %7 = icmp eq i64 %.lobit2, %.lobit
  %or.cond = or i1 %6, %7
  br i1 %or.cond, label %ret, label %overflow

overflow:
  tail call void @ereport(...)

ret:
  ret i64 %5
}
```

Automatic code generation

int.cpp   LLVM C++ API that generates int.bc

```
Function* define_int8pl(Module *mod) {

Function* func_int8pl = Function::Create(..., /*Name=*/"int8pl", mod);

// Block (entry)
Instruction* ptr_1 = GetElementPtrInst::Create(NULL, fcinfo, 0, entry);
LoadInst* int64_2 = new LoadInst(ptr_1, "", false, entry);
Instruction* ptr_3 = GetElementPtrInst::Create(NULL, fcinfo, 1, entry);
LoadInst* int64_4 = new LoadInst(ptr_4, "", false, entry);
BinaryOperator* int64_5 = BinaryOperator::Create(Add, int64_2, int64_4,
entry);
BinaryOperator* lobit = BinaryOperator::Create(LShr, int64_2, 63,
".lobit", entry);
BinaryOperator* lobit1 = BinaryOperator::Create(LShr, int64_4, 63,
".lobit1", entry);
ICmpInst* int1_6 = new ICmpInst(*entry, ICMP_NE, lobit, lobit1);
BinaryOperator* lobit2 = BinaryOperator::Create(LShr, int64_5, 63,
".lobit2", entry);
ICmpInst* int1_7 = new ICmpInst(*entry, ICMP_EQ, lobit2, lobit);
BinaryOperator* int1_or_cond = BinaryOperator::Create(Or, int1_6, int1_7,
"or.cond", entry);
BranchInst::Create(ret, overflow, int1_or_cond, entry);

// Block (overflow)
CallInst* void_err = CallInst::Create(func_erreport, void, overflow);

// Block (ret)
ReturnInst::Create(mod->getContext(), int64_5, ret);

return func_int8pl;
}
```

CPPBackend

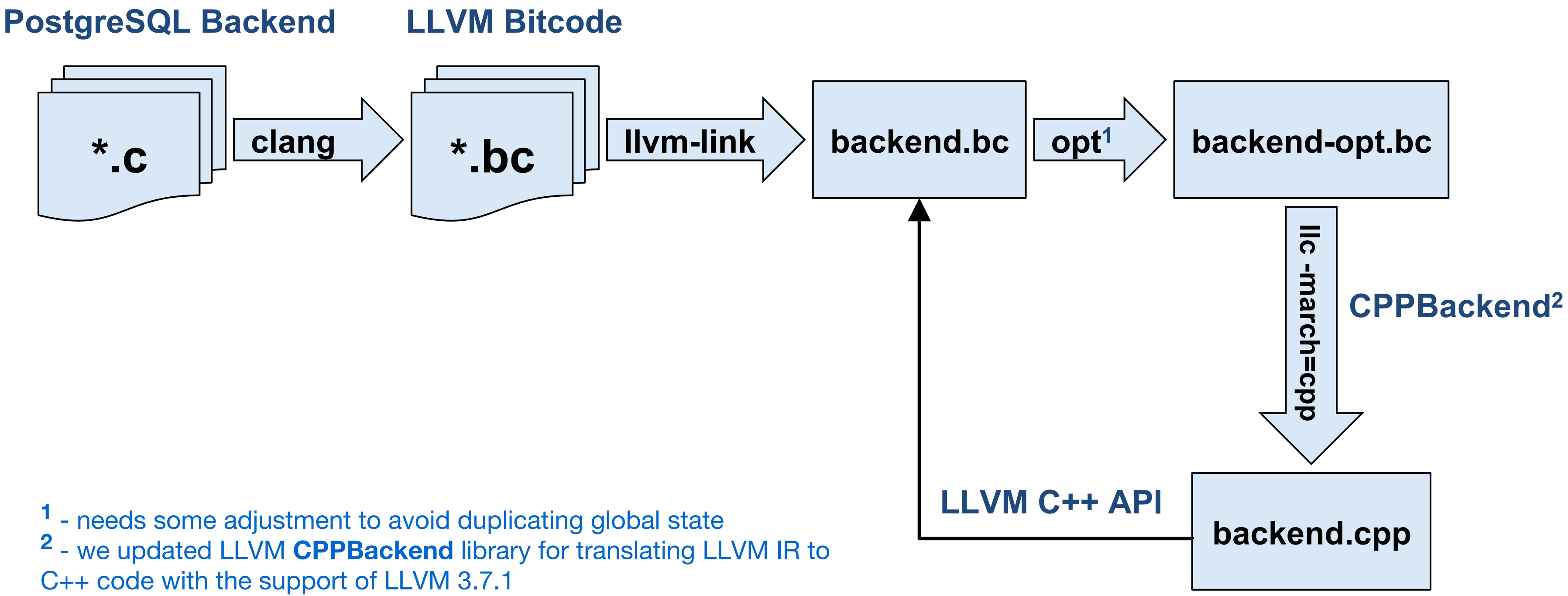
int.bc   LLVM IR

```
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
entry:
  %1 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
  %4 = load i64, i64* %3
  %5 = add nsw i64 %4, %2
  %.lobit = lshr i64 %2, 63
  %.lobit1 = lshr i64 %4, 63
  %6 = icmp ne i64 %.lobit, %.lobit1
  %.lobit2 = lshr i64 %5, 31
  %7 = icmp eq i64 %.lobit2, %.lobit
  %or.cond = or i1 %6, %7
  br i1 %or.cond, label %ret, label %overflow

overflow:
  tail call void @ereport(...)

ret:
  ret i64 %5
}
```

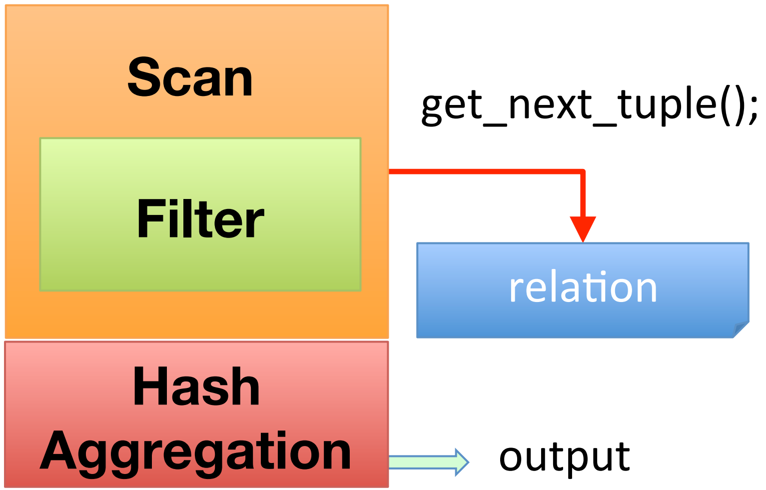
PostgreSQL backend functions to LLVM IR precompilation



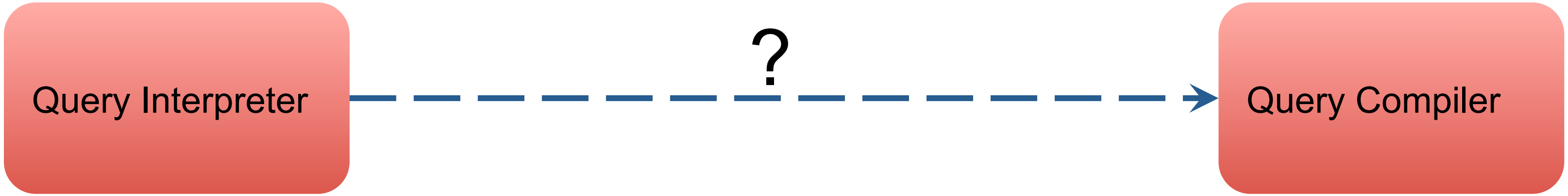
[1] - needs some adjustment to avoid duplicating global state
[2] - we updated LLVM **CPPBackend** library for translating LLVM IR to C++ code with the support of LLVM 3.7.1

Semi-automatic implementation in LLVM

- Semi-automatic implementation of Postgres nodes in LLVM C API
  - When traversing the plan tree, operation in a node isn't executed, instead, it generates a corresponding LLVM IR.
- Getting rid of “Volcano-style” iterative model
- Calls to precompiled backend functions
- Getting rid of the indirect calls to get_next(), use function inlining in LLVM
- Achieve up to **5x** speedup on TPC-H Q1 compared to original PostgreSQL interpreter.



Query Compiler Generation (1)



existing piece of technology
20+ years of development effort

what we want

1. Ideally, Query Compiler is derived from Query Interpreter fully automatically.
2. Can use precompilation technique and specialize Query Interpreter source code for the query at hand.

Query Compiler Generation (2)

- Optimize functions called with constant arguments.

```
declare void @ExecutePlan(%struct.PlanState* %planstate)

define void @ExecutePlan.1() {
        call void @ExecutePlan(i64 3735927486 to %struct.PlanState*)
        ret void
}
```

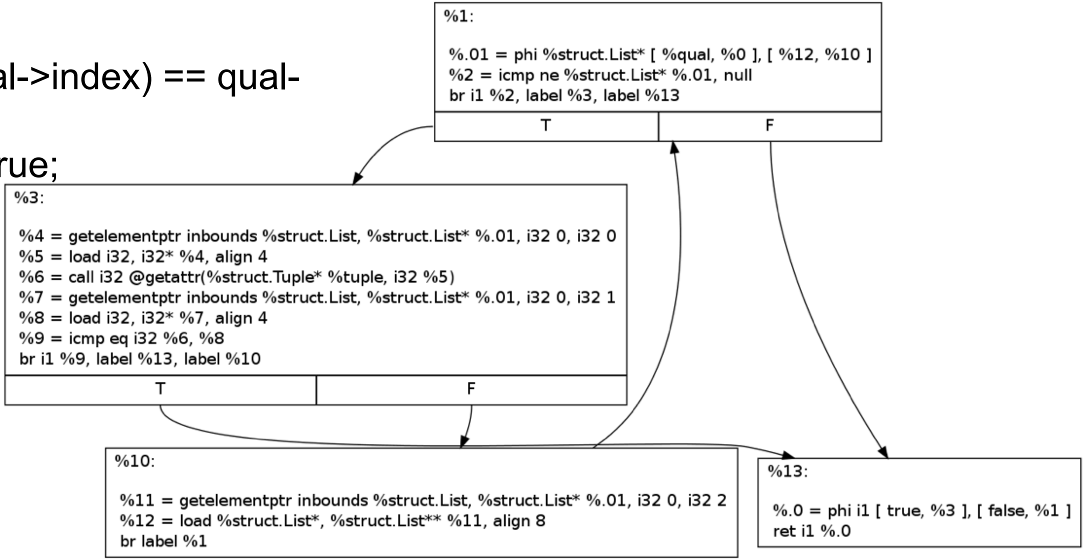
- Need a recursive function pass, with some kind of constant folding / SCCP.
- Challenges:
  - ➢ Need support for tracking memory in order to replace loads from Query Execution Plan with data being loaded.
  - ➢ Need support for CFG restructuring, such as unrolling compile-time loops (very common in Query Interpreter code).

Example (1)

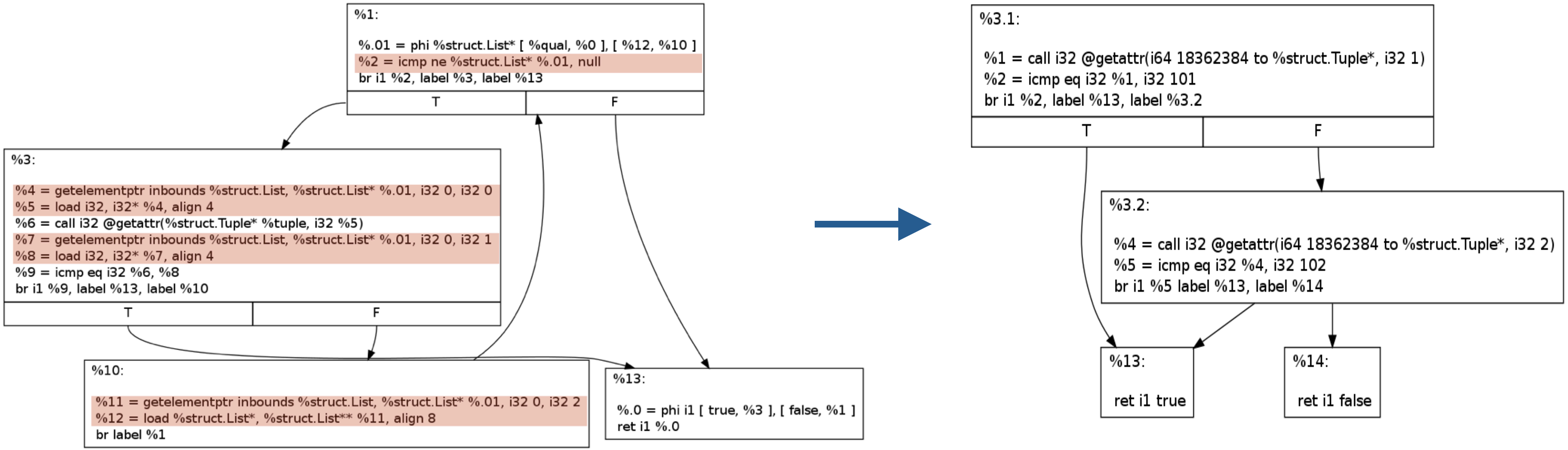
Function CheckQual called with constant arguments:

```
bool
CheckQual (Tuple *tuple, List *qual)
{
        for (; qual; qual = qual->next) {
                if (getattr(tuple, qual->index) == qual-
>value) {
                        return true;
                }
        }
        return false;
}
```

Function CFG:



Example (2)



%1:
%.01 = phi %struct.List* [%qual, %0], [%12, %10]
%2 = icmp ne %struct.List* %.01, null
br i1 %2, label %3, label %13
T
F
%3:
%4 = getelementptr inbounds %struct.List, %struct.List* %.01, i32 0, i32 0
%5 = load i32, i32* %4, align 4
%6 = call i32 @getattr(%struct.Tuple* %tuple, i32 %5)
%7 = getelementptr inbounds %struct.List, %struct.List* %.01, i32 0, i32 1
%8 = load i32, i32* %7, align 4
%9 = icmp eq i32 %6, %8
br i1 %9, label %13, label %10
T
F
%10:
%11 = getelementptr inbounds %struct.List, %struct.List* %.01, i32 0, i32 2
%12 = load %struct.List*, %struct.List** %11, align 8
br label %1
%13:
%.0 = phi i1 [true, %3], [false, %1]
ret i1 %.0
%3.1:
%1 = call i32 @getattr(i64 18362384 to %struct.Tuple*, i32 1)
%2 = icmp eq i32 %1, i32 101
br i1 %2, label %13, label %3.2
T
F
%3.2:
%4 = call i32 @getattr(i64 18362384 to %struct.Tuple*, i32 2)
%5 = icmp eq i32 %4, i32 102
br i1 %5 label %13, label %14
%13:
ret i1 true
%14:
ret i1 false

Semi-Automatic vs. Automatic Approach

1. Precise control over execution model & optimizations.
2. Need to reimplement PostgreSQL execution engine. Huge development & maintenance costs.

1. Limited to Volcano execution model employed by the current PostgreSQL executor.
2. Everything is supported by default. No feature lag.
3. Opportunities to tweak & hand-optimize parts of query executor without having to rewrite it all.
4. Modest development & maintenance effort.

Results (for semi-automatic method)

- PostgreSQL 9.6 beta2
- Database: 100GB (on RamDisk storage)
- CPU: Intel Xeon

Semi-automatic

TPC-H	**Q1**	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q17	Q19	Q20	Q22
Support	**yes**	**partial**	**yes**	**partial**	**partial**	**partial**	**partial**	**partial**	**partial**	**partial**	**partial**	**partial**	**yes**	**partial**	**partial**	**yes**	**yes**	**partial**	**yes**
PG, sec	431,81	22,90	212,06	45,05	255,74	112,52	98,41	41,36	180,78	173,71	11,46	228,55	252,1	127,36	249,93	163,56	9,03	39,2	16,47
JIT, sec	100,52	25,35	103,38	30,01	224,4	36,71	71,39	41,49	152,18	92,97	11,08	131,25	175,9	44,43	161,82	100,4	7,07	37,01	15,29
X times	**4,30**	**0,90**	**2,05**	1,50	1,14	**3,07**	1,38	1,00	1,19	1,87	1,03	1,74	1,43	**2,87**	1,54	1,63	1,28	1,06	1,08

- DECIMAL types in all tables changed to DOUBLE PRECISION and CHAR(1) to ENUM
- **Partial** means successful run with **disabled** BITMAPHEAPSCAN, MATERIAL, MERGE JOIN
- Not yet supported - Q16, Q18, Q21

Conclusion

- Developed PostgreSQL extension for dynamic compilation of SQL-queries using LLVM JIT.
- Developed a tool for automatic compilation of PostgreSQL backend files into C++ code that uses LLVM C++ API and allows to generate LLVM bitcode of backend functions while processing queries.

- Results:
- Semi-automatic:
  - Speedup by ~7 times on simple synthetic tests
  - Speedup by ~5 times on TPC-H Q1
- Automatic:
  - Currently, speedup by 10% on simple synthetic tests

Future work

- Implement on LLVM all types of nodes.
- Testing on TPC-* and other benchmarks, profiling, search of places to optimize.
- Parallelism:
    - Parallel compilation.
- More code to JIT (extensions, access methods, etc.)
- Preparing for release in Open Source, interaction with the PostgreSQL Community.

Thank you!

Questions, comments, feedback:
dm@ispras.ru