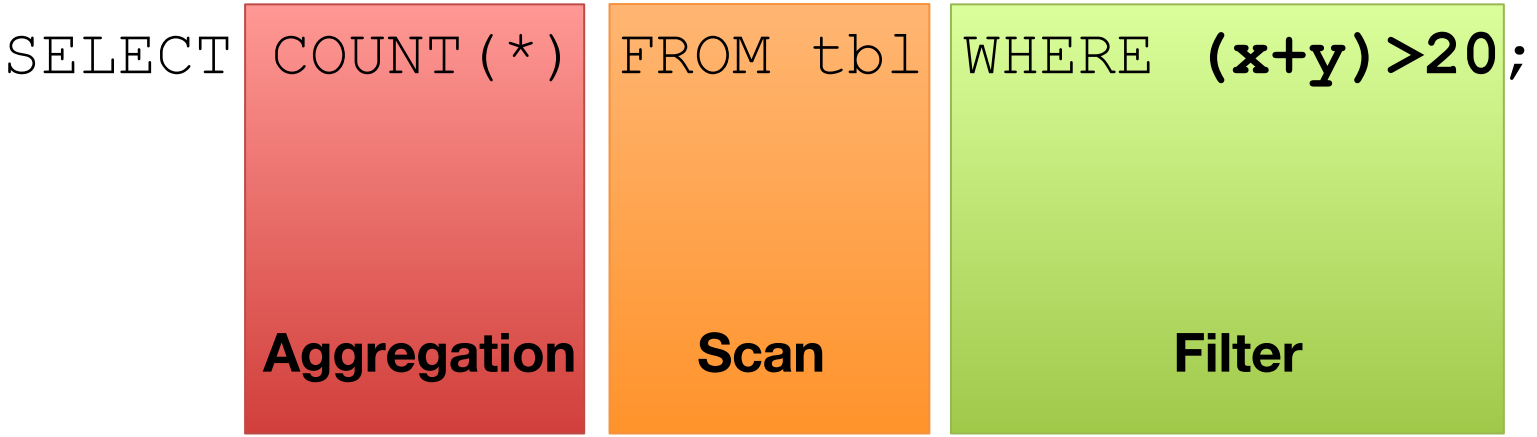
PostgreSQL Acceleration

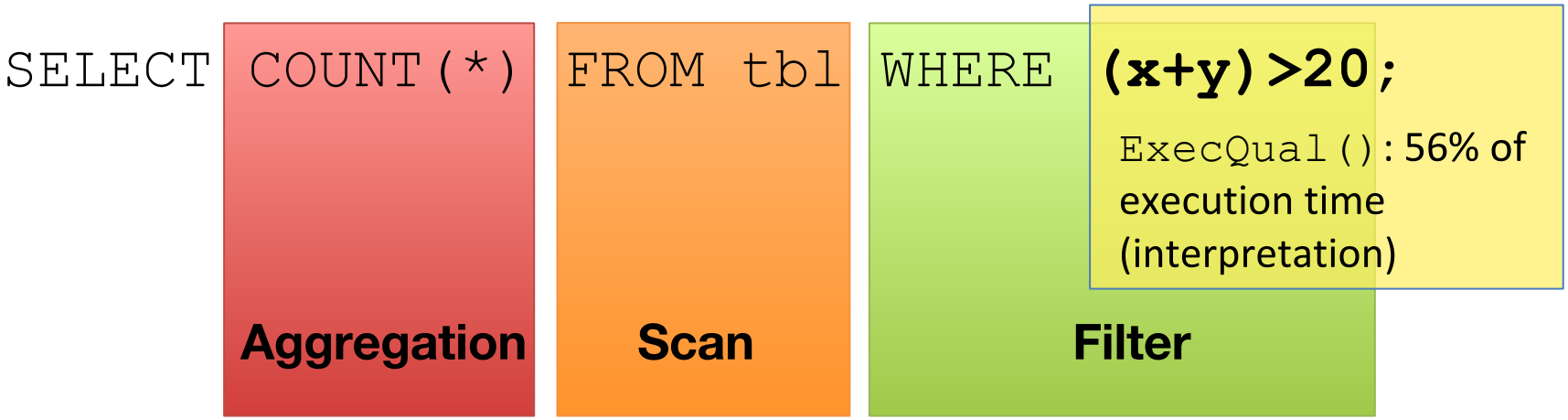
ISP RAS

- 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 JIT; for the first stage – compiling expressions in WHERE clause

Example of query optimization



Example of query optimization



LLVM

ISP RAS

- LLVM (Low Level Virtual Machine) – compiler infrastructure designed for program compilation and optimization
  - Platform-independent internal representation (LLVM bitcode)
  - Wide set of optimizations
  - Code generation for popular platforms (x86, x86_64, ARM, MIPS, …)
  - Well suited for building a JIT-compiler: the dynamic library with an API for generating LLVM bitcode, optimizing and compiling into machine code via just-in-time compilation
  - License: UIUC (permissive BSD-like)
  - Relatively simple code, easy to understand

Using LLVM JIT - popular trend

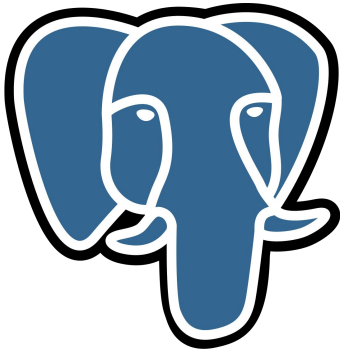
ISP RAS

- Pyston (Python, Dropbox)
- HHVM (PHP & Hack, Facebook)
- LLILC (MSIL, .NET Foundation)
- Julia (Julia, community)

- JavaScript:
  - JavaScriptCore in WebKit (JavaScript, Apple)
  - LLV8 – adding LLVM as a new level of compilation in Google V8 compiler (JavaScript, ISP RAS)

What if we add LLVM JIT to the PostgreSQL?

ISP RAS

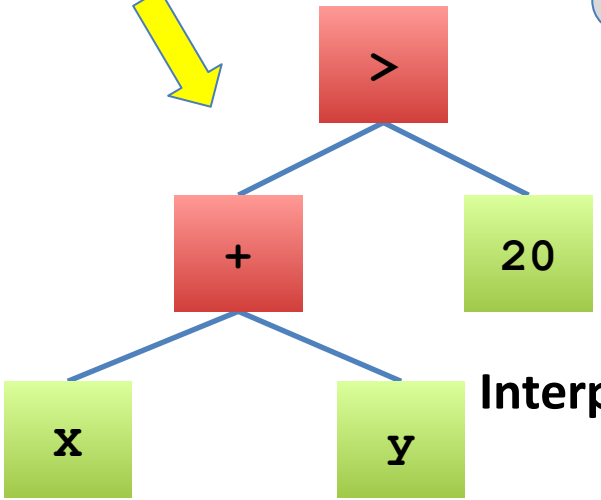


=

Expression evaluation

ISP RAS

(x+y)>20



LLVM bitcode generator

LLVM bitcode

```
define i32 @where_expr(i32 %x, i32 %y) #0 {
entry:
  %add = add nsw i32 %y, %x
  %cmp = icmp sgt i32 %add, 20
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}
```

(overflow checks omitted for simplicity)

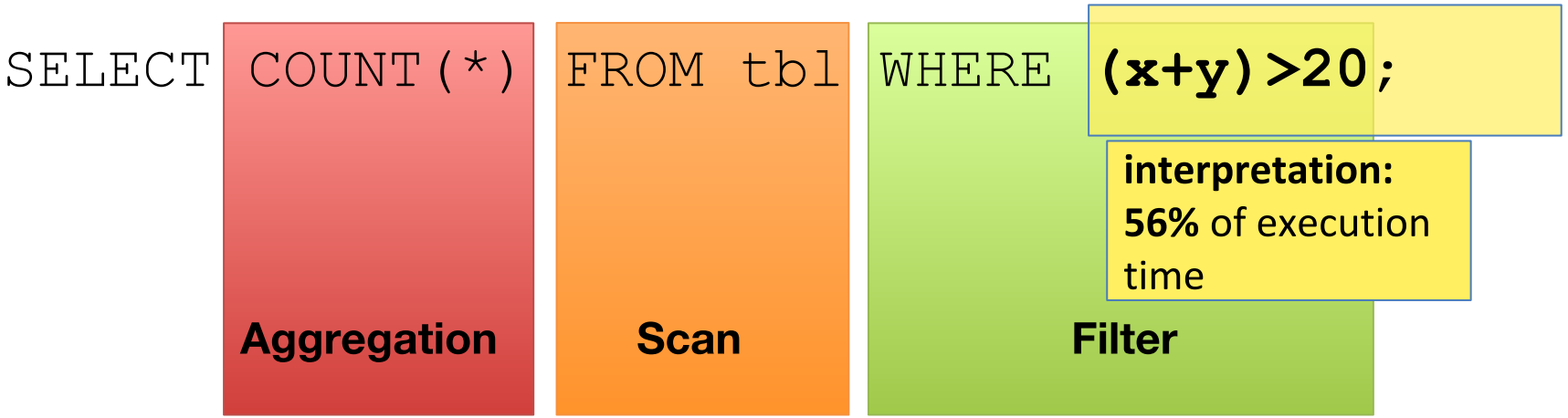
LLVM MCJIT

Interpretation vs **Optimized binary code**

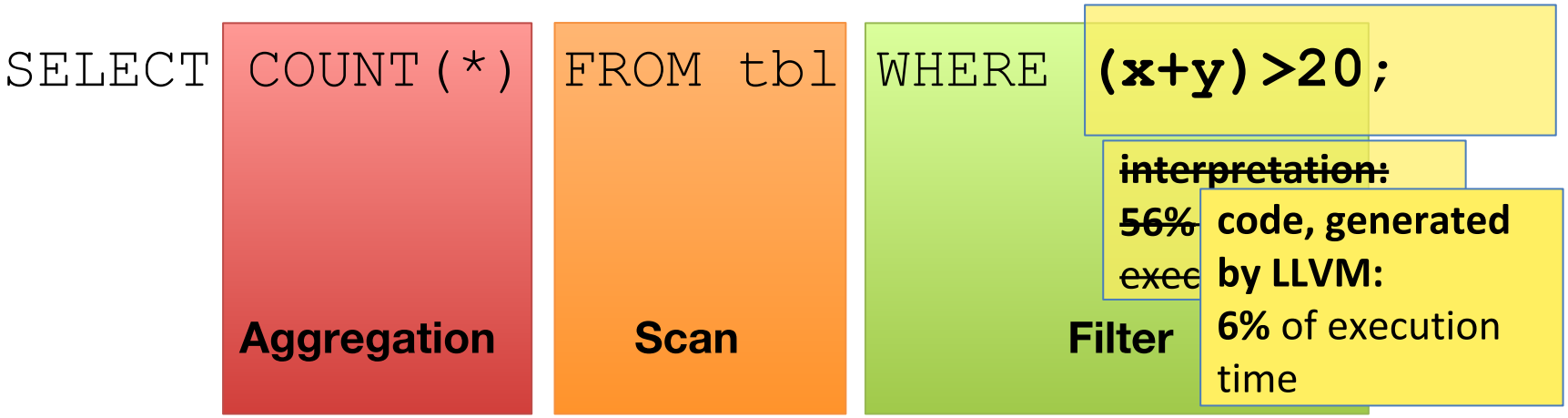
```
foo:
        addl    %esi, %edi
        cmpl    $20, %edi
        setg    %al
        movzbl  %al, %eax
        retq
```

~10 times faster

Example of query optimization



Example of query optimization



=> Acceleration of query execution by 2 times

Related work (1)

ISP RAS

1. T. Neuman. Efficiently Compiling Efficient Query Plans for Modern Hardware. Proceedings of the VLDB Endowment, Vol. 4, No. 9, 2011.
2. PGStrom
   - PostgreSQL extension (Custom scan)
   - Query execution on GPU (Cuda)
3. Axle Project
   - OpenCL

Related work (2)

ISP RAS

4. Vitesse DB
   - Proprietary commercial database based on PostgreSQL 9.4.7
   - Implements optimizations similar to those in [1]:
     - Whole query in one JIT procedure on LLVM
     - Compilation of expressions using LLVM
       - Speedup by 2-8 times on Q1, in average ~3 times on TPC-H
   - Parallel Query Execution and Column Store
     - Speedup by 108/180 times on Q1 with LLVM JIT compilation using row/column store

1st step: JIT expressions only

ISP RAS

- PostgreSQL extension (Custom Scan)
  - ➢ For SeqScan changed filtration:
    - Added compilation of expression tree on LLVM
    - Execution of optimized code for filtering each tuple
    - Supports basic operations for int, float, date (about 100 from ~2000)
- Result: speedup by 2 times on simple synthetic benchmarks
  - removing indirect calls for each operation
  - constants in query => values in the instruction
  - optimization of expressions by means of LLVM

Support of operations in LLVM

ISP RAS

- When traversing the expression tree, operation in a node isn't executed, instead, it generates a corresponding LLVM bitcode.
- It is necessary to implement code generation for all operations of all types supported in expressions (about 2000 functions in total).
  1. Manual implementation of all operations in LLVM C API
     - A lot of tedious work, it is easy to make a mistake, difficult to support when Postgres code changes.
  2. Automatic code generation
     - Pre-compilation of the source code of Postgres backend operations to LLVM bitcode: *src/backend/*.c* => *.bc using Clang compiler.
     - Automatic translation of compiled .bc files into C++ source files that will make calls against the LLVM C++ API to build the same bitcode as the input. This allows to generate bitcode of appropriate backend operations while processing queries in Postgres.
       - The appropriate tool of LLVM is not supported from LLVM 3.4.

Automatic code generation

ISP RAS

int.c PostgreSQL backend file

```
Datum
int4pl(PG_FUNCTION_ARGS)
{
    int32       arg1 = PG_GETARG_INT32(0);
    int32       arg2 = PG_GETARG_INT32(1);
    int32       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_INT32(result);
}
```

Clang

int.bc LLVM Bitcode

```
define i64 @int4pl(%struct.FunctionCallInfoData* %fcinfo) {
  %1 = getelementptr %struct.FunctionCallInfoData,       %struct.
FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = trunc i64 %2 to i32
  %4 = getelementptr %struct.FunctionCallInfoData, %struct.
FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
  %5 = load i64, i64* %4
  %6 = trunc i64 %5 to i32
  %7 = add nsw i32 %6, %3
  %.lobit = lshr i32 %3, 31
  %.lobit1 = lshr i32 %6, 31
  %8 = icmp ne i32 %.lobit, %.lobit1
  %.lobit2 = lshr i32 %7, 31
  %9 = icmp eq i32 %.lobit2, %.lobit
  %or.cond = or i1 %8, %9
  br i1 %or.cond, label %ret, label %overflow

; <label>:overflow
  tail call void @erreport()

; <label>:ret
  %10 = zext i32 %7 to i64
  ret i64 %18
}
```

Clang 3.7 - compiler frontend for LLVM.

Automatic code generation

ISP RAS

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

```
Function* define_int4pl(Module *mod) {

Function* func_int4pl = Function::Create(..., /*Name=*/"int4pl", mod);

// Block (entry)
Instruction* ptr_1 = GetElementPtrInst::Create(NULL, fcinfo, 0, "", entry);
LoadInst* int64_2 = new LoadInst(ptr_1, "", false, entry);
CastInst* int32_3 = new TruncInst(int64_2, IntegerType::get(..., 32), "", entry);
Instruction* ptr_4 = GetElementPtrInst::Create(NULL, fcinfo, 1, "", entry);
LoadInst* int64_5 = new LoadInst(ptr_4, "", false, entry);
CastInst* int32_6 = new TruncInst(int64_5, IntegerType::get(..., 32), "", entry);
BinaryOperator* int32_7 = BinaryOperator::Create(Instruction::Add, int32_6,
int32_3, "", entry);
BinaryOperator* lobit = BinaryOperator::Create(Instruction::LShr, int32_3, 31, ".
lobit", entry);
BinaryOperator* lobit1 = BinaryOperator::Create(Instruction::LShr, int32_6, 31, ".
lobit1", entry);
ICmpInst* int1_8 = new ICmpInst(*entry, ICmpInst::ICMP_NE, lobit, lobit1, "");
BinaryOperator* lobit2 = BinaryOperator::Create(Instruction::LShr, int32_7, 31, ".
lobit2", entry);
ICmpInst* int1_9 = new ICmpInst(*entry, ICmpInst::ICMP_EQ, lobit2, lobit, "");
BinaryOperator* int1_or_cond = BinaryOperator::Create(Instruction::Or, int1_8,
int1_9, "or.cond", entry);
BranchInst::Create(ret, overflow, int1_or_cond, entry);

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

// Block (ret)
CastInst* int64_10 = new ZExtInst(int32_7, IntegerType::get(..., 64), "", ret);
ReturnInst::Create(mod->getContext(), int64_10, ret);

return func_int4pl;
}
```

llvm2cpp ⟸

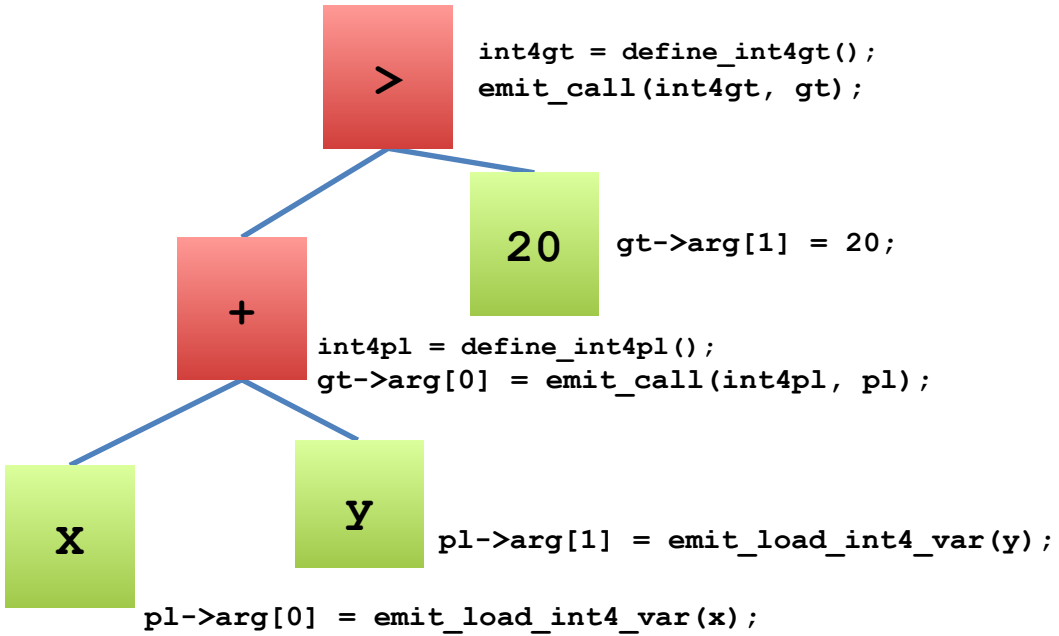
int.bc — LLVM Bitcode

```
define i64 @int4pl(%struct.FunctionCallInfoData* %fcinfo) {
  %1 = getelementptr %struct.FunctionCallInfoData,       %struct.
FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = trunc i64 %2 to i32
  %4 = getelementptr %struct.FunctionCallInfoData, %struct.
FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 1
  %5 = load i64, i64* %4
  %6 = trunc i64 %5 to i32
  %7 = add nsw i32 %6, %3
  %.lobit = lshr i32 %3, 31
  %.lobit1 = lshr i32 %6, 31
  %8 = icmp ne i32 %.lobit, %.lobit1
  %.lobit2 = lshr i32 %7, 31
  %9 = icmp eq i32 %.lobit2, %.lobit
  %or.cond = or i1 %8, %9
  br i1 %or.cond, label %ret, label %overflow

; <label>:overflow
  tail call void @erreport()

; <label>:ret
  %10 = zext i32 %7 to i64
  ret i64 %18
}
```

llvm2cpp - updated LLVM CPPBackend library for converting LLVM bitcode to C++ code with the support of LLVM 3.7.

Code generation for expressions

```
FunctionCallInfo gt, pl;
```



1. *define_int4gt, define_int4pl*
   - LLVM C++ API code generators
   - Automatically generated at build time.
2. LLVM bitcode is generated in a post-order traversal of the expression tree.
3. For each operation node, a corresponding function (*define_ **) is called to generate code for the operation.
4. For each variable node, attribute offset is precomputed.
5. For each constant node, constant value is inlined.

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;
```

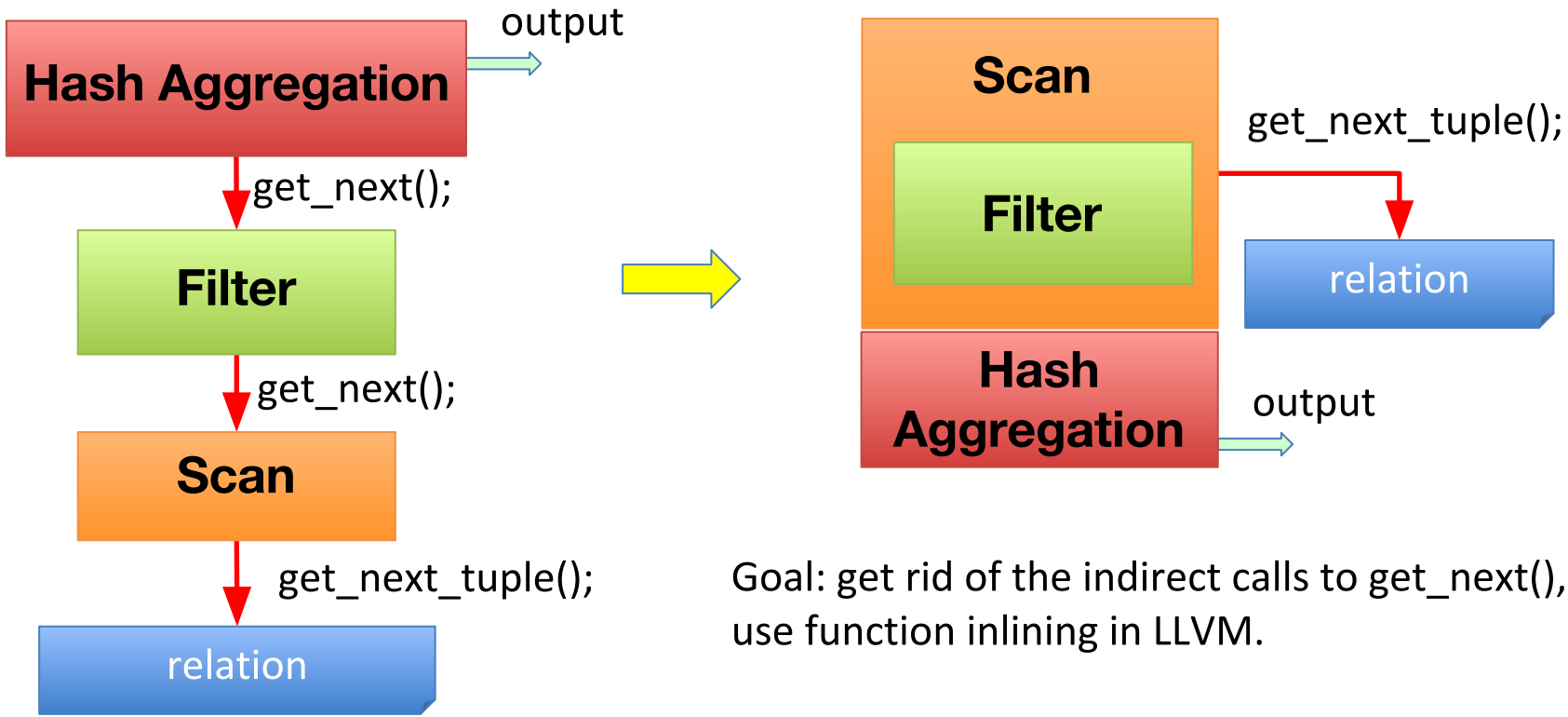
Function	TPC-H Q1	TPC-H Q2	TPC-H Q3	TPC-H Q6	TPC-H Q22	Average on TPC-H
ExecQual	6%	14%	32%	3%	72%	25%
ExecAgg	75%	-	1%	1%	2%	16%
SeqNext	6%	1%	33%	-	13%	17%
IndexNext	-	57%	-	-	19%	38%
BitmapHeapNext	-	-	-	85%	-	85%

2nd step: implementation of Scan Aggregation and Join on LLVM

ISP RAS

- PostgreSQL extension (Executor hook)
  - ➢ Implementation of Scan, Aggregation and Join on LLVM
    - Supported SeqScan, IndexScan, IndexOnlyScan, Aggregation (Plain, Hash), NestLoop, HashJoin.
  - ➢ Getting rid of “Volcano-Style” iterative model
    - Implementation of Join, Aggregation, Scan and Filter on LLVM as one function(cycle/loop), Scan performs as the outer loop.
- Result: speedup on **TPC-H Q1** by **3.3x** times on 100GB table.

Getting rid of “Volcano-style” iterative model



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

Example: code generation for HashAgg

Postgres iterative model

```
TupleTableSlot *
ExecScan(...)
{
...
    for (;;)
    {
        TupleTableSlot *slot = ExecScanFetch(node, accessMtd, recheckMtd);

        /*
         * check that the current tuple satisfies the qual-clause
         */
        if (!qual || ExecQual(qual, econtext, false))
        {
            /*
             * Form a projection tuple
             */
            if (projInfo)
            {

                return ExecProject(projInfo, &isDone);
            }
            else
                return slot;
        }
    }
...
}

void
agg_fill_hash_table(...)
{
...
    for (;;)
    {
            TupleTableSlot *outerslot = ExecScan(aggstate);
            if (TupIsNull(outerslot))
               break;

            entry = lookup_hash_entry(aggstate, outerslot);
            advance_aggregates(aggstate, entry->pergroup);
    }
...
}
```

manually

HashAgg and Scan in one JIT procedure (Linear)

```
LLVMValueRef
scan_consume_codegen(...)
{
...
    /* Check filter condition */
    if (qual)
    {
        LLVMValueRef pred = GenerateQual(qual, econtext, scanslot, NULL, NULL);
        LLVMBuildCondBr(pred, filtered_bb, continue_bb);
    }

    /* filtered_bb: form a projection tuple */
    if (projInfo)
    {
        GenerateProject(projInfo, scanslot, NULL, NULL);
    }

    /* Produce slot */
    stop = LLVMBuildCall(agg_consume_f, NULL, 0, "stop");
    LLVMBuildRet(stop);

    /* continue_bb */
...
    return scan_consume_f;
}

LLVMValueRef
hash_agg_consume_codegen(...)
{
...
    /* entry */
    entry = LLVMBuildCall(lookup_hash_entry_f, NULL, 0, "entry");
    entry = LLVMBuildPointerCast(entry, LLVMPointerType(LLVMInt8Type(), 0), "entry");
    ret = LLVMConstInt(LLVMInt32Type(), offsetof(AggHashEntryData, pergroup), 0);
    pergroup = LLVMBuildGEP(entry, &ret, 1, "pergroup");
    pergroup = LLVMBuildPointerCast(pergroup, AggStatePerGroupDataPointerType(), "pergroup");
    LLVMBuildCall(advance_aggregates_f, &pergroup, 1, "");
...
    return agg_consume_f;
}
```

Results

ISP RAS

- PostgreSQL 9.6 trunk as of 16.10.2015
- Database: 100GB (on RamDisk storage)
- CPU: Intel Xeon

TPC-H	Q1	Q3	Q4	Q5	Q6	Q7	Q8	Q10	Q11	Q12	Q13	Q14	Q15	Q17	Q19
Support	yes	yes	partial	partial	partial	partial	partial	partial	partial	partial	yes	partial	partial	partial	yes
PG, sec	441,41	217,13	403,91	208,84	119,18	108,14	47,32	199,63	11,22	215,07	250,06	128,56	304,77	11,89	9,93
JIT, sec	132,1	164,67	361,54	154,43	57,61	99,44	42,68	136,84	10,58	162,44	184,59	74,52	245,04	8,65	6,14
X times	**3,34**	1,32	1,12	1,35	**2,07**	1,09	1,11	**1,46**	1,06	1,32	1,35	**1,73**	1,24	1,37	**1,62**
%	**70,07%**	24,16%	10,49%	26,05%	**51,66%**	8,05%	9,81%	**31,45%**	5,70%	24,47%	26,18%	**42,03%**	19,60%	27,25%	**38,17%**

- 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 - Q2, Q9, Q16, Q18, Q20-22

Conclusion

ISP RAS

- Developed PostgreSQL extension for dynamic compilation of SQL-queries using LLVM JIT. Implemented phases:
  - Filtration (support compilation of all operations of all types)
  - Scan (SeqScan, IndexScan, IndexOnlyScan)
  - Aggregation (Plain, Hash)
  - Join (NestLoop, HashJoin)
- 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:
  - Speedup by ~2 times on simple synthetic tests when JIT only expressions of WHERE clause.
  - Speedup by ~3.3 times on TPC-H Q1.

Future work

ISP RAS

- Implement on LLVM all types of scanning, aggregation, join, sorting (including in one cycle)
  - BitmapHeapScan, Agg Sorted, Merge Join, Material etc.
- Testing on TPC-* and other benchmarks, profiling, search of places to optimize.
- Parallelism:
  - Implementation of parallel scanning and aggregation on LLVM.
  - Parallel compilation.
- More code to JIT (extensions, access methods, etc.)
- Preparing for release in Open Source, interaction with the PostgreSQL Community.