ИСП РАН Институт системного программирования Российской академии наук

Динамическая компиляция SQL-запросов в PostgreSQL с использованием LLVM JIT

Д.М. Мельник, Р.А. Бучацкий, Р.А. Жуйков, Е.Ю. Шарыгин
Институт системного программирования Российской академии наук
(ИСП РАН)
17 марта 2017

Dynamic Compilation of SQL Queries in PostgreSQL Using LLVM JIT

Dmitry Melnik, Ruben Buchatskiy, Roman Zhuykov, Eugene Sharygin
Institute for System Programming of the Russian Academy of Sciences
(ISP RAS)
March 17, 2017

Motivational Example

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

Aggregation | Scan | Filter

interpreter:
56% of execution time

Motivational Example

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

Aggregation

Scan

Filter

~~interpreter:~~
~~56%~~
~~time~~

LLVM-generated code:
6% of execution time

=> Speedup query execution 2 times

Project Goals

- Speed up PostgreSQL for computationally intensive SQL-queries
- What exactly we want to speed up?
  - Complex queries where performance "bottleneck" is CPU rather than disk (primarily analytics, but not limited to)
  - Optimize performance for TPC-H benchmark
- How to achieve speedup?
  - Dynamically compile queries to native code using LLVM JIT

Related Work

- Neumann T., Efficiently Compiling Efficient Query Plans for Modern Hardware. Proceedings of the VLDB Endowment, Vol. 4, No. 9, 2011.
- Vitesse DB:
  - Proprietary database based on PostgreSQL
  - JIT compiling expressions as well as execution plan
  - Speedup up to 8x on TPC-H Q1
- Butterstein D., Grust T., Precision Performance Surgery for PostgreSQL – LLVM-based Expression Compilation, Just in Time. VLDB 2016.
  - JIT compiling expressions for Filter and Aggregation
  - Speedup up to 37% on TPC-H
- New expression interpreter for Postgres (WIP by Andres Freund):
  - Changes tree walker based interpreter to more effective one
  - Also adds LLVM JIT for expressions (~20% speedup on TPC-H Q1)

LLVM

- LLVM (Low Level Virtual Machine) – compiler infrastructure designed for program compilation and optimization
  - Platform-independent internal representation (LLVM IR)
  - Wide set of compiler optimizations
  - Code generation for popular platforms (x86/ x86_64, ARM32/64, MIPS, …)
  - Well suited for building a JIT-compiler: the dynamic library with an API for generating LLVM IR, 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 is a popular trend

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

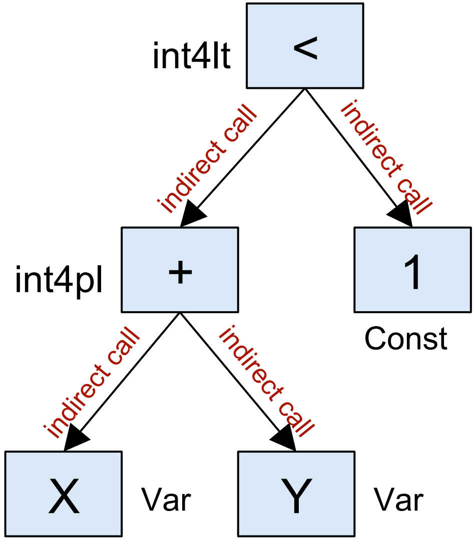
- JavaScript:
  - JavaScriptCore in WebKit (Apple) – Fourth Tier LLVM JIT (FTL JIT), recently replaced by B3 (Apple's custom JIT)
  - LLV8 – LLVM added to Google V8 (open source project by ISP RAS)

- DBMS:
  - MemSQL, Impala
  - … and now PostgreSQL

Adding LLVM JIT to PostgreSQL?

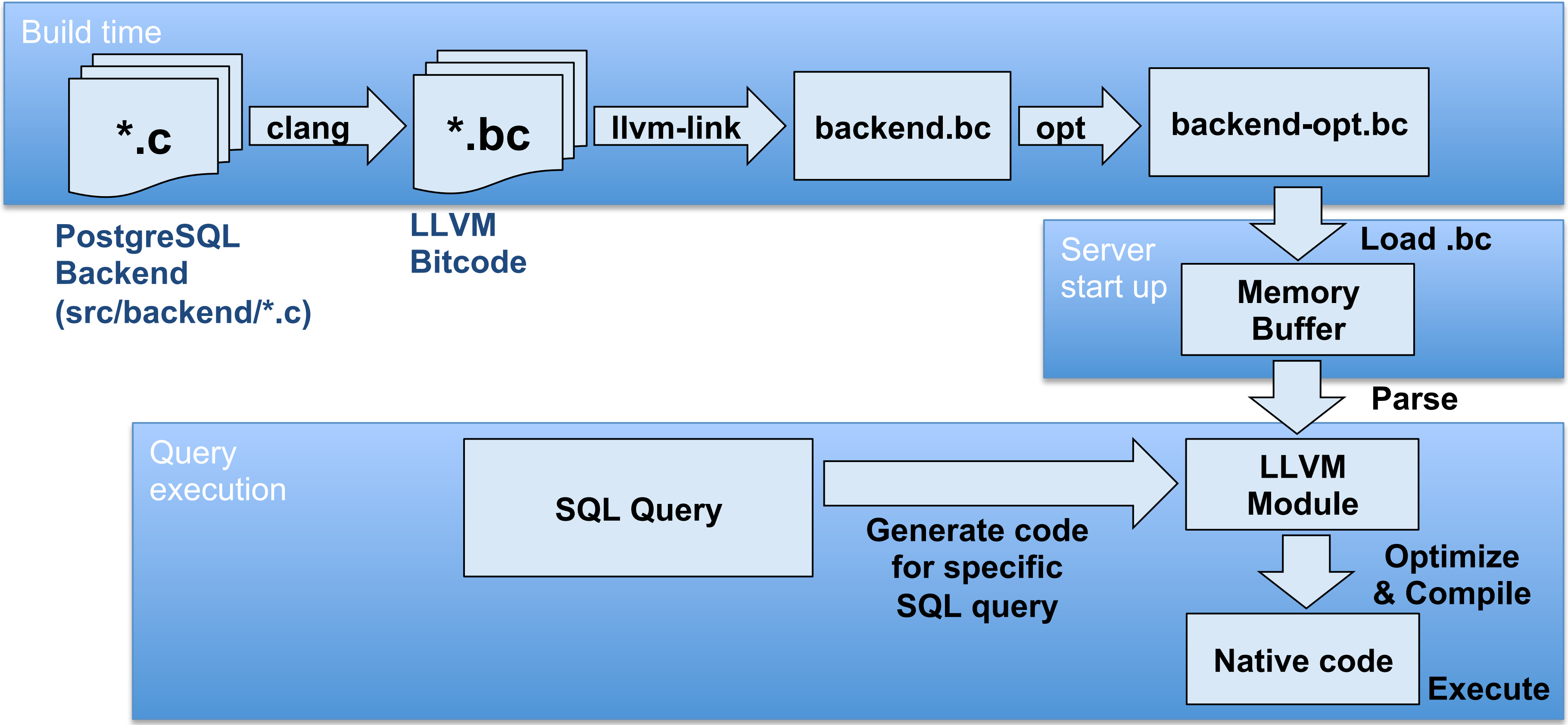
JIT-compiling Expressions

X+Y < 1




```
define i1 @ExecQual() {
  %x = load &X.attr
  %y = load &Y.attr

  %pl = add %x, %y
  %lt = icmp lt %pl, 1

  ret %lt
}
```

Precompiling Postgres Backend Functions



Pre-compiling backend functions

```c
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("bigint out of range")));
  PG_RETURN_INT64(result);
}
```

Clang

```llvm
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
  %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:
  call void @ereport(...)
ret:
  ret i64 %5
}
```

PostgreSQL int8.c

LLVM IR int8.bc

JIT Compilation at Different Levels

TPC-H Q1 speedup ~

Expressions JIT

backend functions

supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):

```
a*a + b*b <= r*r
```

17%

+ supports built-in functions:

```
sqrt(pow(a, 2) + pow(b, 2)) <= r
```

3%

20%

- Based on Postgres 9.6.1
- TPC-H Q1 speedup is 20%
- Expressions JIT is published as open source and available at **github.com/ispras/postgres**

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 '90' day
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

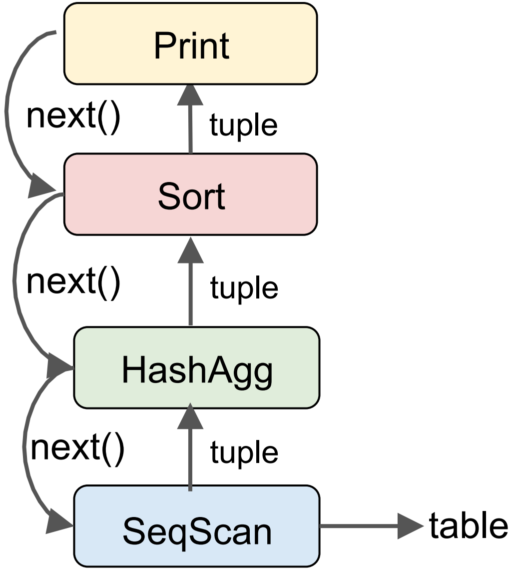
Function	TPC-H Q1	TPC-H Q2	TPC-H Q3	TPC-H Q6	TPC-H Q22
ExecQual	6%	14%	32%	3%	72%
ExecAgg	75%	-	1%	1%	2%
SeqNext	6%	1%	33%	-	13%
IndexNext	-	57%	-	-	13%
BitmapHeapNext	-	-	-	85%	-

Executor JIT Overview

1. Use `ExecutorRun_hook` (before execution start)
2. Check whether all nodes, functions and expressions used in query plan are supported by JIT
   – If the query unsupported, call standard PostgreSQL interpreter
3. Generate LLVM IR for query
4. Compile to native code using LLVM JIT
5. Run code

Plan Execution: Volcano model

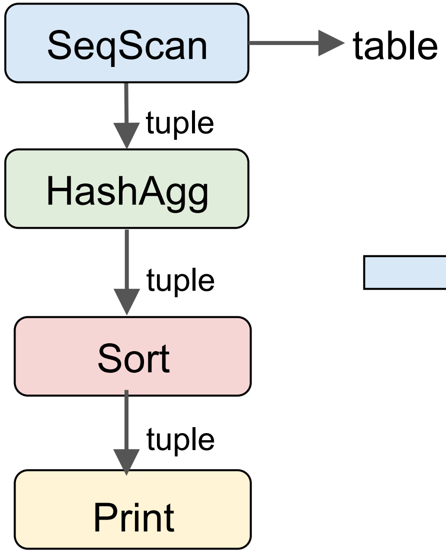
- Graefe G., Volcano— An Extensible and Parallel Query Evaluation System. IEEE TKDE 6 (1), 120-135, 1994
- Each operator is represented as a sequence of tuples, which is accessed by calling method next() (ExecProcNode in Postgres)
- Indirect function call: branch misprediction, inlining is impossible
- Need to store/load internal state between calls to next()




```
select a, sum(b) from tbl
group by a order by a;
```

Plan Execution: push-based model

- Query execution is controlled by the leaf node of Plan tree
- The query is represented as number of loops

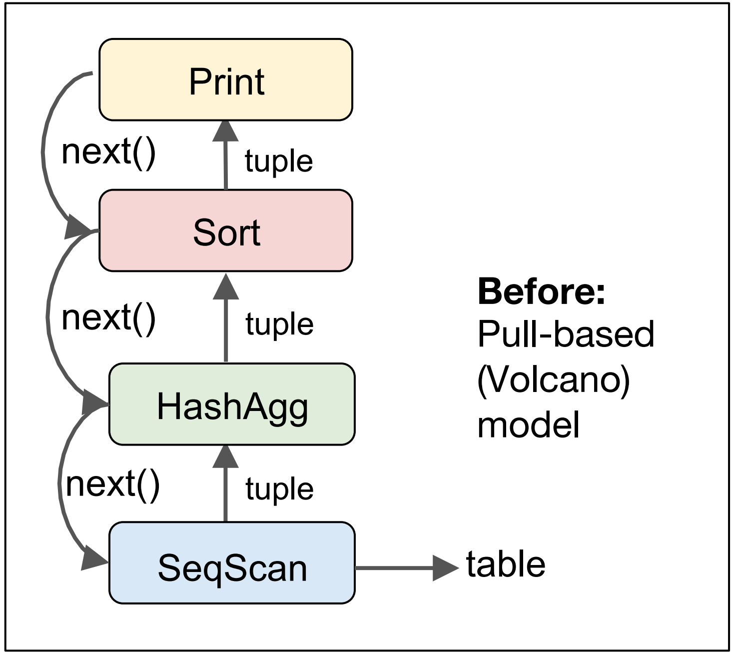


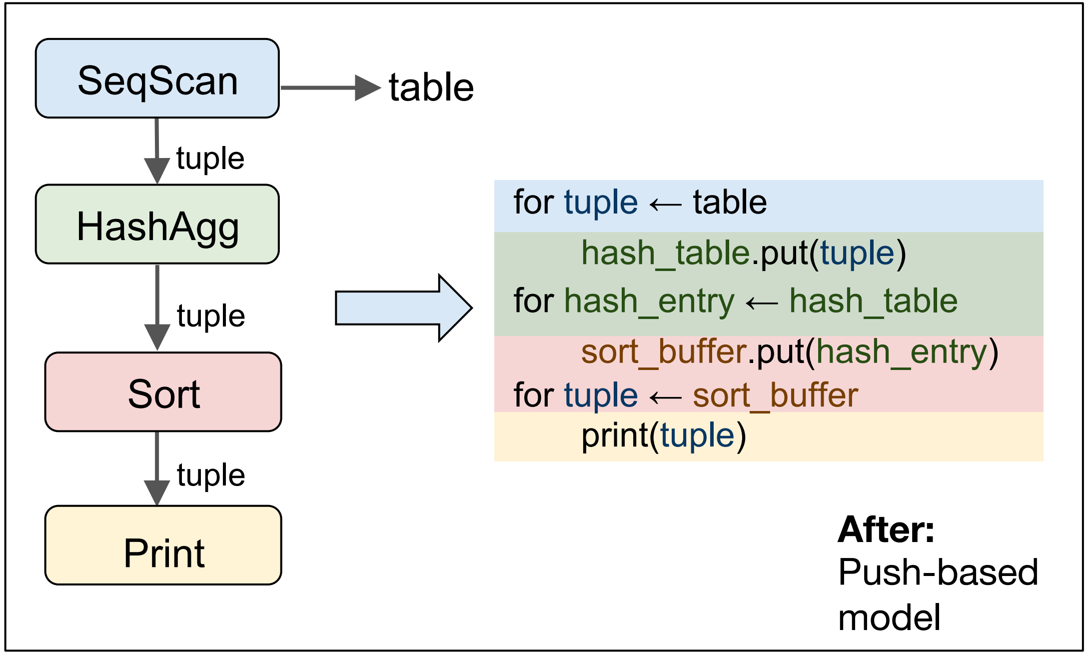

```
for tuple ← table
        hash_table.put(tuple)
for hash_entry ← hash_table
        sort_buffer.put(hash_entry)
for tuple ← sort_buffer
        print(tuple)
```

Changing Execution Model

select <columns> from <table> where <condition> group by <column>
order by <column>;

HashAgg **SeqScan** **Filter** **Sort**



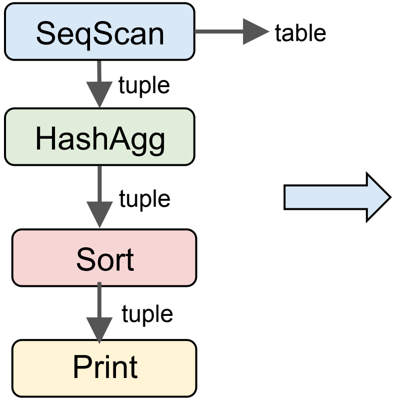


Plan Execution: push-based model

- Generator functions for LLVM C API:

```
LLVMFunction HashAgg(LLVMFunction consume, LLVMFunction finalize)
```

- Generated functions contain calls to consume() for each output tuple and finalize() after processing all tuples




```
SeqScan(c, f)
HashAgg(c, f) = SeqScan(HashAgg.consume(),
                        HashAgg.finalize(c, f))
Sort(c, f) = HashAgg(Sort.consume(),
                     Sort.finalize(c, f))

Print() = Sort(print, null)
```

Plan Execution: push-based model

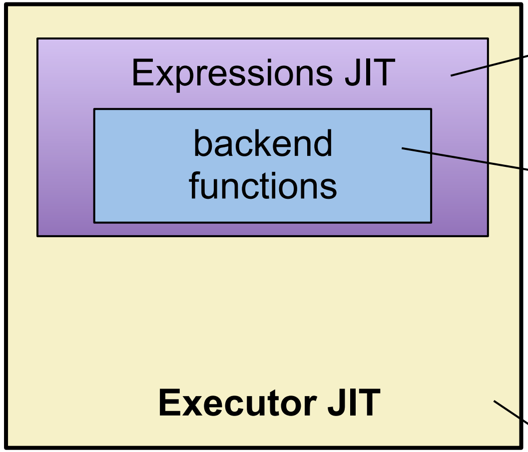
```
llvm.seqscan() {
    for tuple ← table
        llvm.hashagg.consume(tuple)
    llvm.hashagg.finalize()
}
llvm.hashagg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.hashagg.finalize() {
    for hash_entry ← hash_table
        llvm.sort.consume(hash_entry)
    llvm.sort.finalize()
}
llvm.sort.consume(tuple) {
    sort_buffer.put(tuple)
}
llvm.sort.finalize() {
    for tuple ← sort_buffer
        print(tuple)
}
```

inlining (LLVM)

```
main() {
    for tuple ← table
        hash_table.put(tuple)
    for hash_entry ← hash_table
        sort_buffer.put(hash_entry)
    for tuple ← sort_buffer
        print(tuple)
}
```

- No indirect calls
- No need to store internal state

JIT Compilation at Different Levels



supports expressions in Filter (WHERE)
and Aggregation (sum, count, avg, …):

```
a*a + b*b <= r*r
```

+ supports built-in functions:

```
sqrt(pow(a, 2) + pow(b, 2)) <= r
```

+ compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model

TPC-H Q1
speedup ~

17%

3%

20%

JIT-compiling attribute access

- slot_deform_tuple
- Optimize out:
  - attribute number
  - nullability
  - attribute lengths
  - unused attributes

```
for (attnum = 0; attnum < natts; attnum++) {
    Form_pg_attribute thisatt = att[attnum];

    if (att_isnull(attnum, bp)) {
        values[attnum] = (Datum) 0;
        isnull[attnum] = true;
        continue;
    }

    isnull[attnum] = false;

    off = att_align_nominal(off, thisatt->attalign);

    values[attnum] = fetchatt(thisatt, tp + off);

    off = att_addlength_pointer(off, thisatt->attlen,
                                tp + off);
}
```

```
isnull[0] = false;
values[0] = *(int32 *)(tp);
isnull[2] = false;
values[2] = *(int32 *)(tp + 8);
...
```

JIT Compilation at Different Levels

ISP RAS

TPC-H Q1 speedup ~

Executor JIT

Expressions JIT

backend functions

slot_deform_tuple()

supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):

`a*a + b*b <= r*r`

17%

+ supports built-in functions:

`sqrt(pow(a, 2) + pow(b, 2)) <= r`

3%

20%

+ optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes

+ compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model

5.5x times

- Implemented as an extension for Postgres 9.6.2
- TPC-H Q1 speedup is 5.5x times
- Continuing work on Executor JIT
- Compilation time is sufficient for short-running queries

Results for Executor JIT

ISPRAS

- Measured on PostgreSQL 9.6.1, extension with LLVM 4.0 ORC JIT
- Database: 75GB (on RamDisk storage, data folder size ~200GB), shared_buffers = 32GB
- CPU: Intel Xeon E5-2699 v3.

TPC-H-like workload	**Q1**	Q3	Q9	Q13	Q17	Q19	Q22	Q6	Q14	Q15	Q18	Q12	Q10	Q4	Q2	Q20	Q7	Q11	Q5	Q8
PostgreSQL (sec)	283,27	107,38	197,27	134,92	8,62	6,10	12,65	72,94	77,48	139,56	130,48	136,29	98,59	45,02	17,64	79,93	153,69	6,13	342,66	43,08
+with JIT (sec)	52,11	60,26	164,52	95,69	5,99	4,83	9,74	25,83	31,52	63,20	61,74	72,74	56,10	34,93	15,57	75,63	147,44	5,95	338,60	42,89
Compilation	0,81	1,05	1,51	0,93	0,78	0,99	1,12	0,40	0,72	1,04	1,03	0,89	1,25	0,71	2,20	0,96	1,45	1,14	1,24	1,62
Speedup, (times)	**5,44**	**1,78**	1,20	1,41	1,44	1,26	1,30	**2,82**	**2,46**	**2,21**	**2,11**	**1,87**	**1,76**	1,29	1,13	1,06	1,04	1,03	1,01	1,00

- Type DECIMAL changed to DOUBLE PRECISION; CHAR(1) to ENUM.
- Bitmap Heap Scan, Material, Merge Join turned off for queries, marked with yellow; Q16 и Q21 are not yet supported

Saving optimized native code for PREPARED queries

ISP RAS

- When using *generic* plan for prepared statements it's possible to save generated code for the plan and then reuse it, eliminating compilation overhead for OLTP workload
- Slower than our regular JITted code because it can't contain immediate values and absolute addresses for structures allocated at the stage of query initialization (e.g. in ExecutorStart)
- Increased memory footprint because of storing native code along with the plan for PREPARED query

Prototype implementation for saving code

Modified code generation to add needed indirection for simple query:

```
set enable_indexscan = 0; set enable_bitmapscan = 0;
prepare qz as select * from lineitem where l_quantity < 1;
```

Size = 1 GB	Postgres, no JIT	JIT extension	JIT and saving code (first time)	Execute loaded code (next times)
Memory, bytes	6736	8480	8304	
Compile time, ms	-	172	174	0
Exec time, ms	525	156	170	170
Total time, ms	525	328	344	170

Can save ~170ms on compilation, but the query runs ~15ms (9%) slower and takes extra ~1.5Kb memory (in addition to ~6.5Kb for saved plan context)

Using llvm.patchpoint

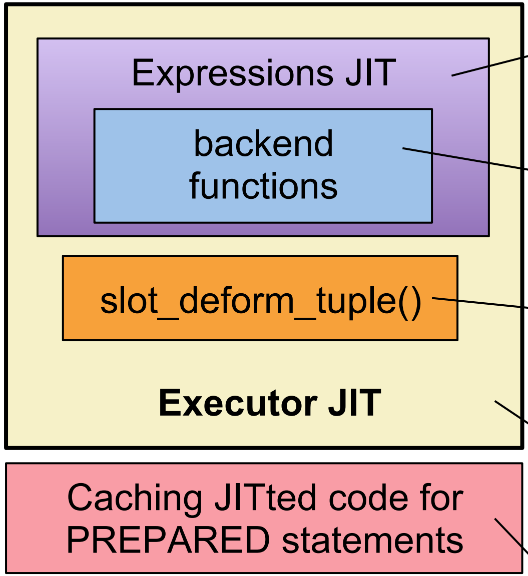
- Now working on more general approach for saving native code for prepared queries using *llvm.patchpoint* to reuse the code with new addresses for structures and possibly new parameter values

- For more complex queries like those in TPC-H native code takes up to 2-5 times more memory than saved generic plan. To store both plan and native code it takes 3-6 times more memory, and on average 4.5 times more (110 Kb instead of 25Kb for single query)

- Possible to save memory by storing only selected (“hottest”) queries, or don’t compile Postgres built-in functions with LLVM, calling them from Postgres binary instead, but this will be slower (up to ~30%)

JIT Compilation at Different Levels

TPC-H Q1 speedup ~



supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):

```
a*a + b*b <= r*r
```

17%

+ supports built-in functions:

```
sqrt(pow(a, 2) + pow(b, 2)) <= r
```

3%

20%

+ optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes

+ compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model

5.5x times

+ Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)

+ eliminates compilation overhead

JIT-compiling Index Creation

- *comparetup_index_btree()* takes ~25-30% of total index creation time
- JIT-compiling *comparetup_index_btree()* and comparators for different types
- Comparators are inlined into *comparetup_index_btree()* during JIT-compilation
- Below are results for creating indexes for 2GB TPC-H-like database

Type	Speedup	Original time, sec	JIT time, sec	Query
(INT, DOUBLE)	7.21%	7.9	7.3	CREATE INDEX i_l_orderkey_quantity ON lineitem (l_orderkey, l_quantity);
INT	5.29%	9.7	9.2	CREATE INDEX i_l_suppkey ON lineitem (l_suppkey);
DOUBLE	15.23%	13.6	11.5	CREATE INDEX i_l_quantity ON lineitem (l_quantity);
VARCHAR	8.95%	16.9	15.4	CREATE INDEX i_l_comment ON lineitem (l_comment);
DATE	7.73%	9.6	8.9	CREATE INDEX i_l_shipdate ON lineitem (l_shipdate);
CHAR	18.59%	21.8	17.7	CREATE INDEX i_l_shipinstruct ON lineitem (l_shipinstruct);

JIT Compilation Everywhere

ISP RAS

TPC-H Q1 speedup ~

Expressions JIT

backend functions

slot_deform_tuple()

Executor JIT

Caching JITted code for PREPARED statements

Index Creation JIT

supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):

`a*a + b*b <= r*r`

17%

+ supports built-in functions:

`sqrt(pow(a, 2) + pow(b, 2)) <= r`

3%

20%

+ optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes

+ compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model

5.5x times

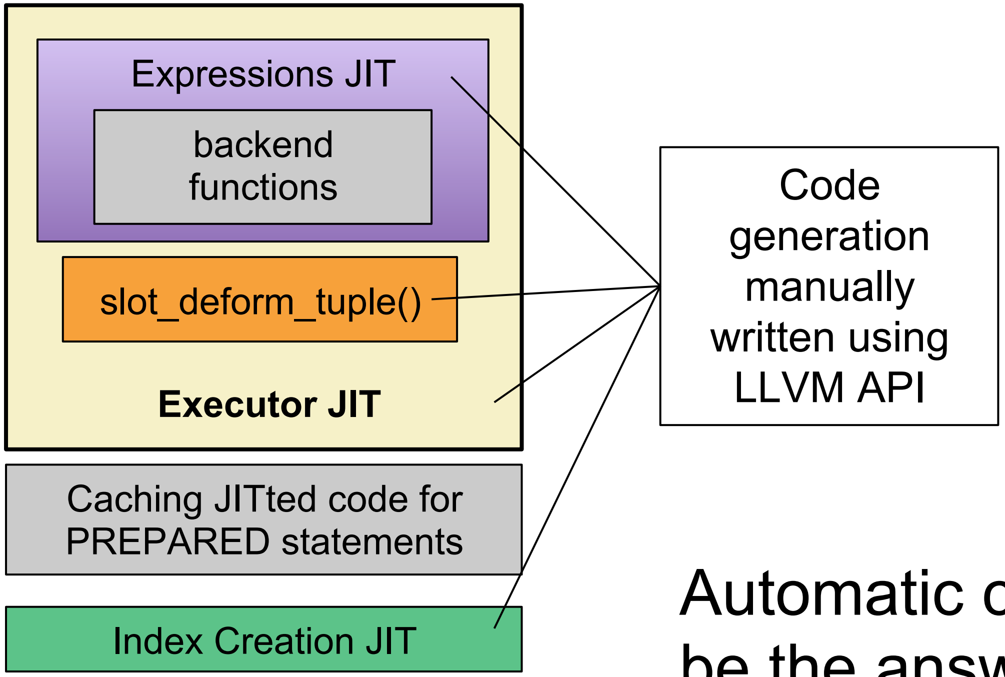
+ Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP)

+ eliminates compilation overhead

+ Compiling creation of *btree* index

5-19%

Automatic JIT generation?



Challenges:

- Same semantics implemented twice
- Maintainability: new features should be added to both interpreter and JIT, tested separately, etc.

Automatic code specialization could be the answer

Run time code specialization

ISP RAS

Program:
```
int foo(int x, int y) {
  return x * y;
}
```

Data:
```
x = 1
```

Code specializer

Program specialized for data:
```
int foo_x1(int y) {
  return 1 * y;
}
```

- For s given SQL query in Postgres:
  - static data (Plan, PlanState, EState) — depend only from SQL query itself, and are invariant during query execution
  - dynamic data (HeapTuple, ItemPointer) — depend both from SQL query and data

Run-time specialization

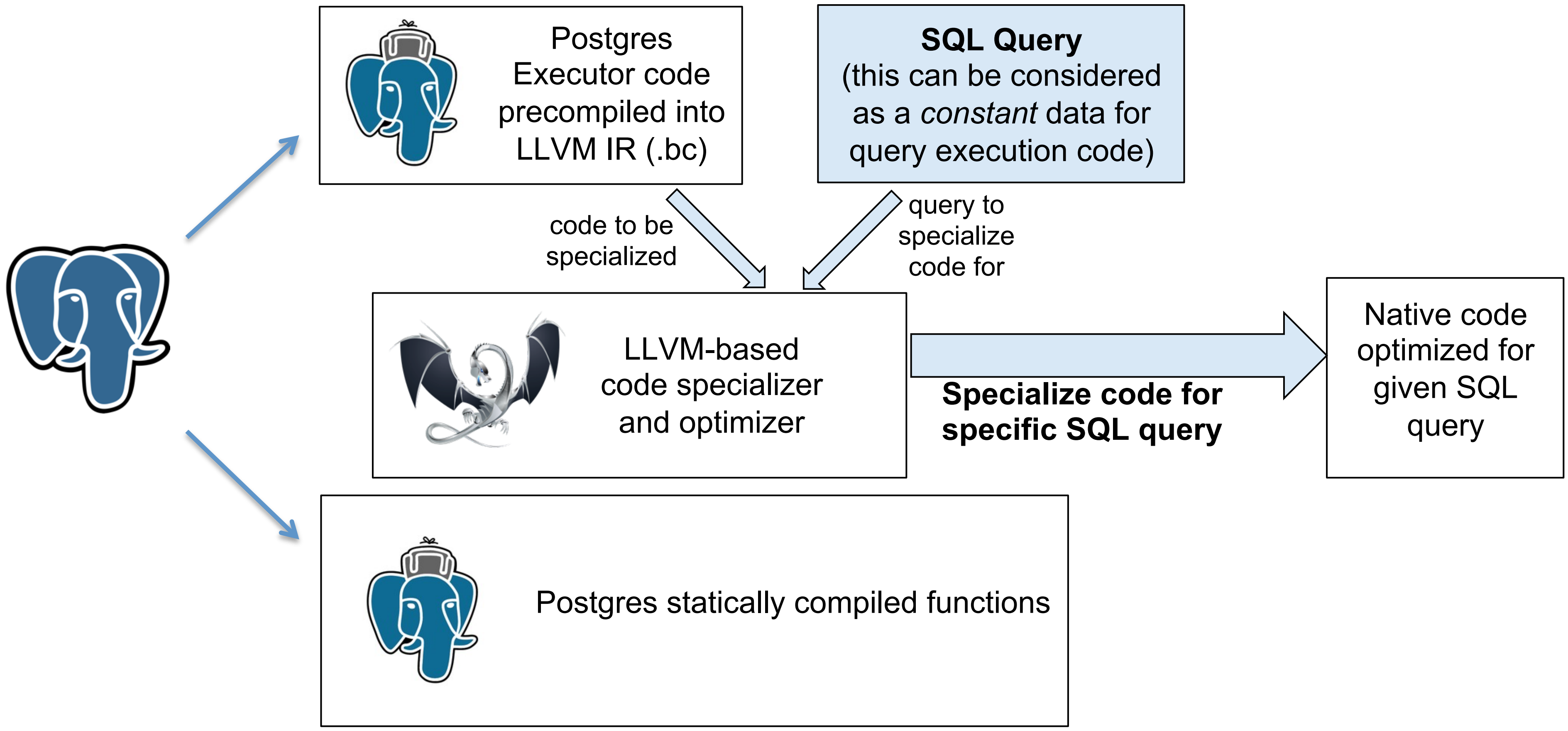
```
for (attnum = 0; attnum < natts; attnum++) {
    Form_pg_attribute thisatt = att[attnum];

    if (att_isnull(attnum, bp)) {
        values[attnum] = (Datum) 0;
        isnull[attnum] = true;
        continue;
    }

    isnull[attnum] = false;

    off = att_align_nominal(off, thisatt->attalign);

    values[attnum] = fetchatt(thisatt, tp + off);

    off = att_addlength_pointer(off, thisatt->attlen,
                                tp + off);
}
```

```
if (att_isnull(0, bp)) {
    values[0] = (Datum) 0;
    isnull[0] = true;
} else {
    isnull[0] = false;
    values[0] = fetchatt(tp);
}
if (att_isnull(1, bp)) {
    values[1] = (Datum) 0;
    isnull[1] = true;
} else {
    isnull[1] = false;
    values[1] = fetchatt(tp + 4);
}
if (att_isnull(2, bp)) {
    values[0] = (Datum) 0;
    isnull[0] = true;
} else {
    isnull[0] = false;
    values[0] = fetchatt(tp + 8);
}
```

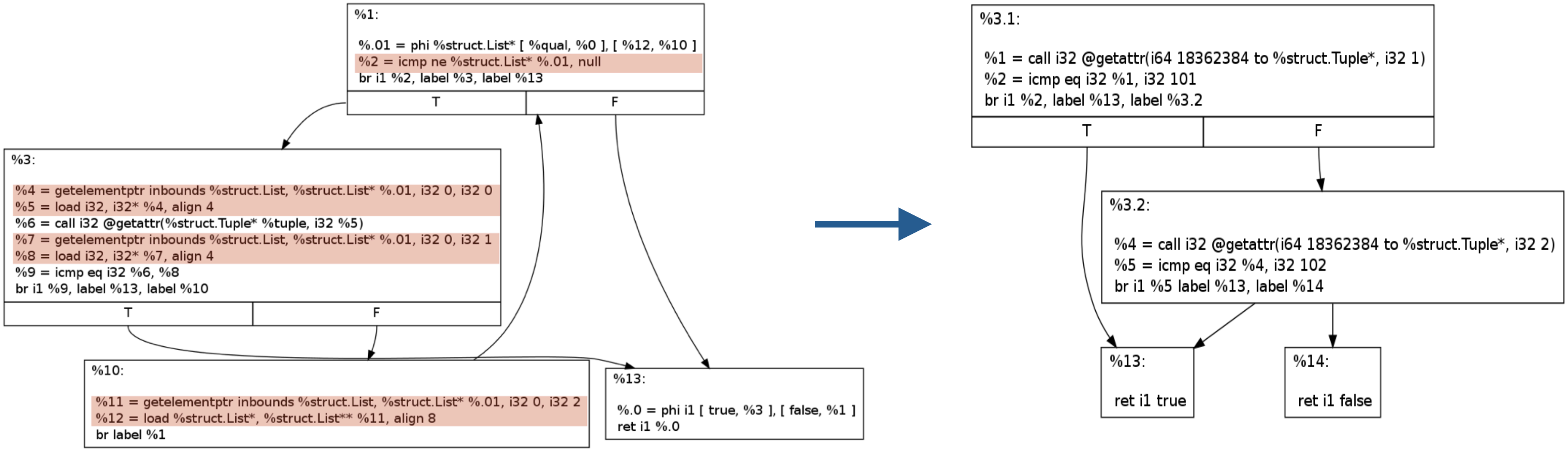
Data available at run-time specialization is marked in RED

- Can unroll loops and pre-compute offsets
- This is actually done in LLVM IR (C shown for readability)

Run time code specialization



Run time code specialization



JIT Compilation Everywhere

ISP RAS

TPC-H Q1 speedup ~

Executor JIT

- **Expressions JIT**: supports expressions in Filter (WHERE) and Aggregation (sum, count, avg, …):
  `a*a + b*b <= r*r` — 17%
  - **backend functions**: + supports built-in functions:
    `sqrt(pow(a, 2) + pow(b, 2)) <= r` — 3%
  - Expressions JIT + backend functions together: **20%**
- **slot_deform_tuple()**: + optimizes fetching attributes from tuple according to current query, i.e. fetching only necessary attributes
- **Executor JIT**: **+ compiles execution plan, i.e. Executor tree nodes (Scan / Aggregation / Join / Sort) manually rewritten using LLVM API; implements *Push* model**

All of the above: **5.5x times**

Caching JITted code for PREPARED statements: + Saving compiled code for PREPARED statements to avoid compile time overhead for short queries (OLTP) — + eliminates compilation overhead

Index Creation JIT: + Compiling creation of *btree* index — **5-19%**

Conclusions

- Expression JIT
  - Open source: github.com/ispras/postgres
  - Speedup up to **20%** on TPC-H
- PostgreSQL Extension JIT (still developing)
  - Speedup up to **5.5 times** on TPC-H
  - Caching native code for PREPARED statements to reduce compilation time
- Index creation JIT
  - Up to 19% speedup
- Developing automatic code specialization
  - Developing Push-model Executor in Postgres (filed as GSoC project)
- Feedback is needed!
  - If you have a workload that you think can benefit from JIT, we'll be happy to test and tune for it
  - We're open for collaboration!

Thank you!

Questions, comments, feedback:

dm@ispras.ru