JIT-Compiling SQL Queries in PostgreSQL Using LLVM

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

* dm@ispras.ru

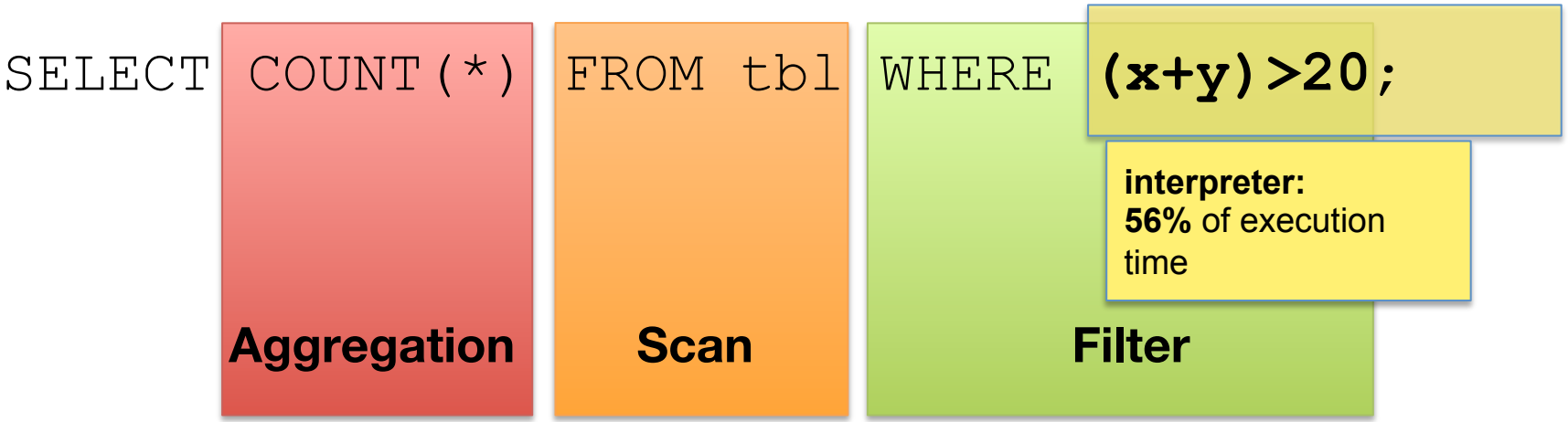
May 26, 2017

Agenda

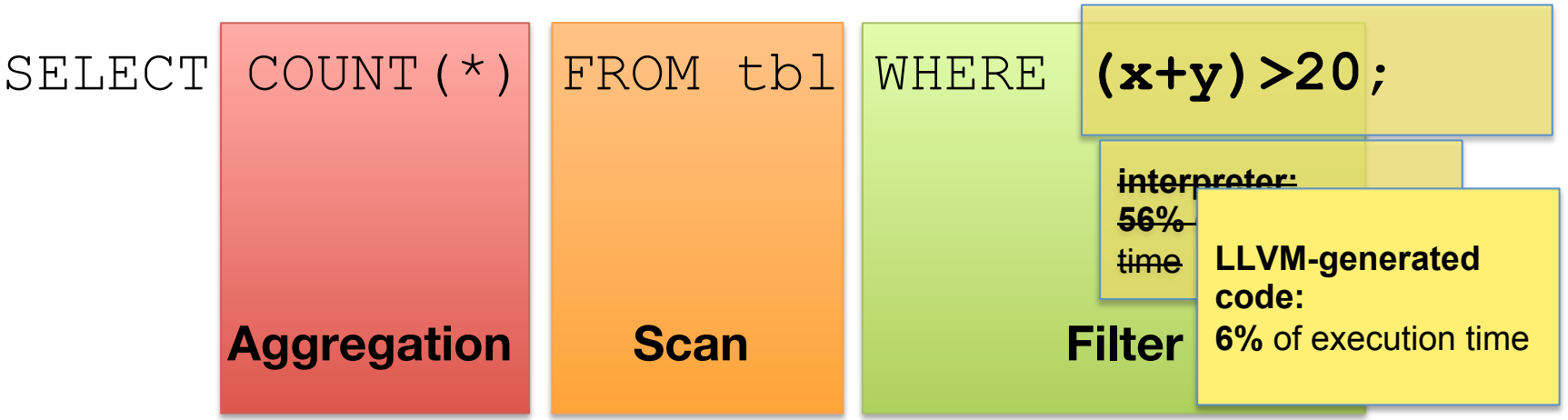
ISP RAS

- Expression JIT
- Full Executor JIT
- Caching JITted code for PREPARED statements
- Index creation JIT
- Experimental
  - Run time Executor code specialization for a given query
  - Switching original PostgreSQL Executor from pull to push model

Motivational Example



Motivational Example



=> Speedup query execution 2 times

ISP RAS

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

Ahead of Time vs JIT Compilation

- What are the benefits for JIT compilation? Can't we just build PostgreSQL code with LLVM with -O3 and all its fancy optimizations?
  - At the time of query execution we have extra information we don't have at PostgreSQL build time
  - We know DB schema, affected tables, attributes, execution plan and filtering conditions
  - With JIT we generate native code specifically to execute given query
  - Such code can be much simpler and effective

Related Work

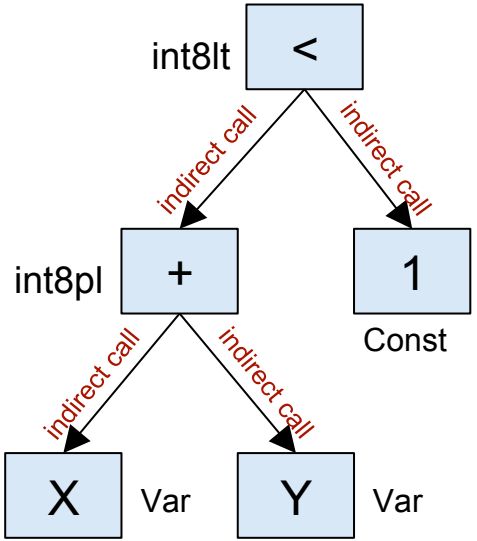
- Neumann T., Efficiently Compiling Efficient Query Plans for Modern Hardware. Proceedings of the VLDB Endowment, Vol. 4, No. 9, 2011.
- Vitesse DB:
  - Proprietary database based on PostgreSQL 9.5.3
  - JIT compiling expressions as well as execution plan
  - Speedup up to 8x on TPC-H Q1
- Butterstein 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 (by Andres Freund, PostgreSQL 10, master):
  - Changes tree walker based interpreter to more effective one
  - Also adds LLVM JIT for expressions

ISP RAS

Adding LLVM JIT to PostgreSQL?

JIT-compiling Expressions

X+Y < 1




```
indirect call ExecEvalFunc()
  P = indirect call ExecEvalFunc()
    X = indirect call ExecEvalVar()
    Y = indirect call ExecEvalVar()
    return int8pl(X, Y)
  C = indirect call ExecEvalConst(1)
  return int8lt(P, C)
```

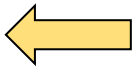
PostgreSQL

JIT-compiling Expressions

X+Y < 1

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

  %pl = llvm.int8pl(%x, %y)
  %lt = llvm.int8lt(%pl, 1)

  ret %lt
}
```

LLVM IR



```
indirect call ExecEvalFunc()
  P = indirect call ExecEvalFunc()
    X = indirect call ExecEvalVar()
    Y = indirect call ExecEvalVar()
    return int8pl(X, Y)
  C = indirect call ExecEvalConst(1)
  return int8lt(P, C)
```

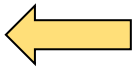
PostgreSQL

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
}
```

LLVM IR (inlining)



```
indirect call ExecEvalFunc()
  P = indirect call ExecEvalFunc()
    X = indirect call ExecEvalVar()
    Y = indirect call ExecEvalVar()
    return int8pl(X, Y)
  C = indirect call ExecEvalConst(1)
  return int8lt(P, C)
```

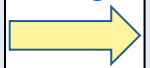
PostgreSQL

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);
}
```

PostgreSQL int8.c

Clang

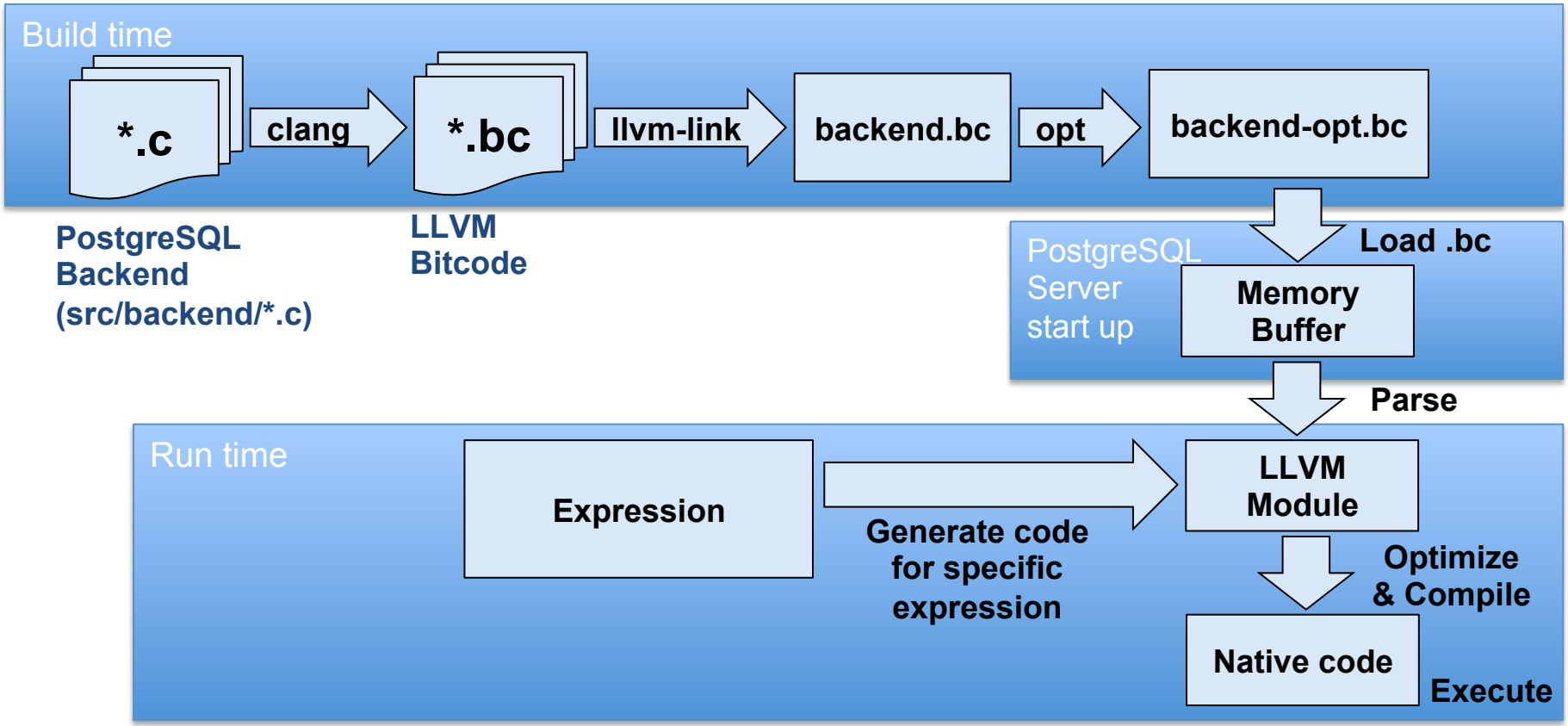


```llvm
define i64 @int8pl(%struct.FunctionCallInfoData* %fcinfo) {
  %1 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 6, i64 0
  %2 = load i64, i64* %1
  %3 = getelementptr %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %fcinfo, i64 0, i32 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
}
```

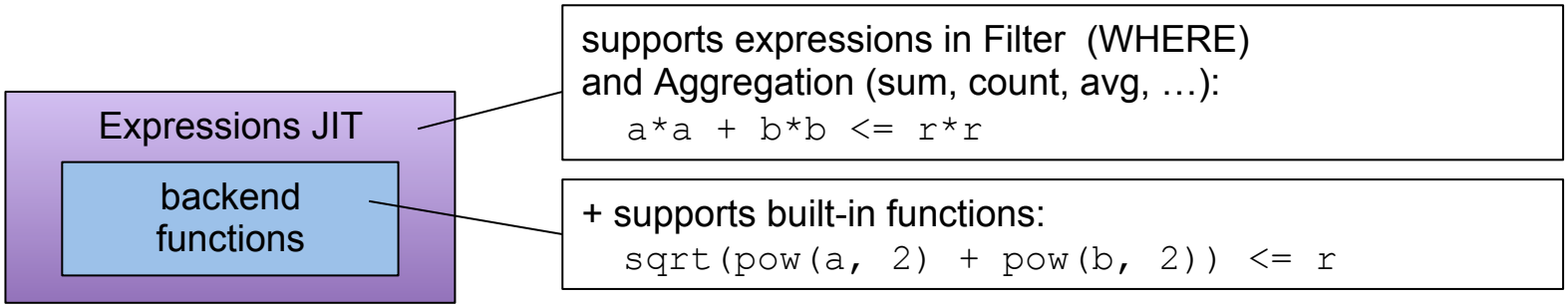
LLVM IR int8.bc

Pre-compiling PostgreSQL Backend Functions

ISP RAS



JIT Compilation at Different Levels



TPC-H Q1 speedup ~

17%

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

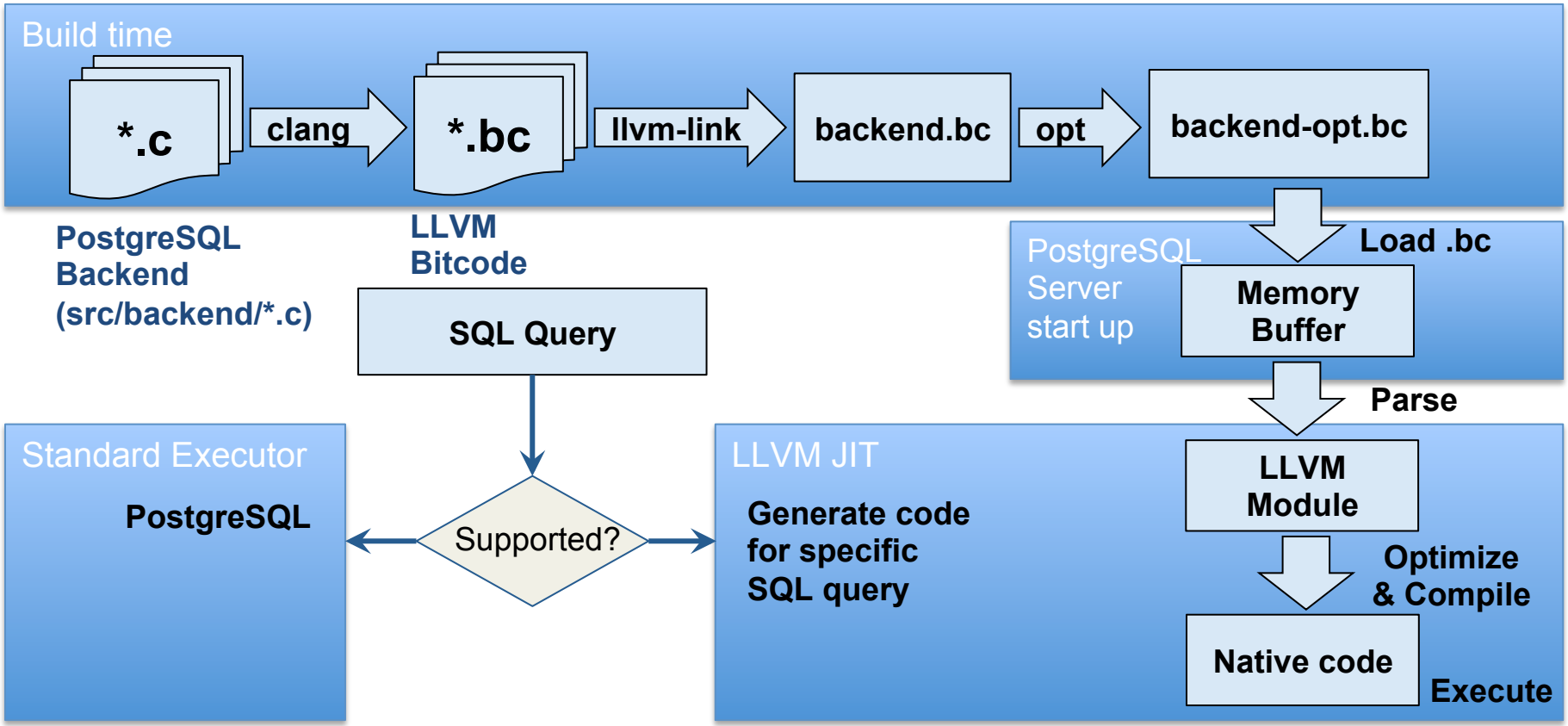
TPC-H Q1:

```
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

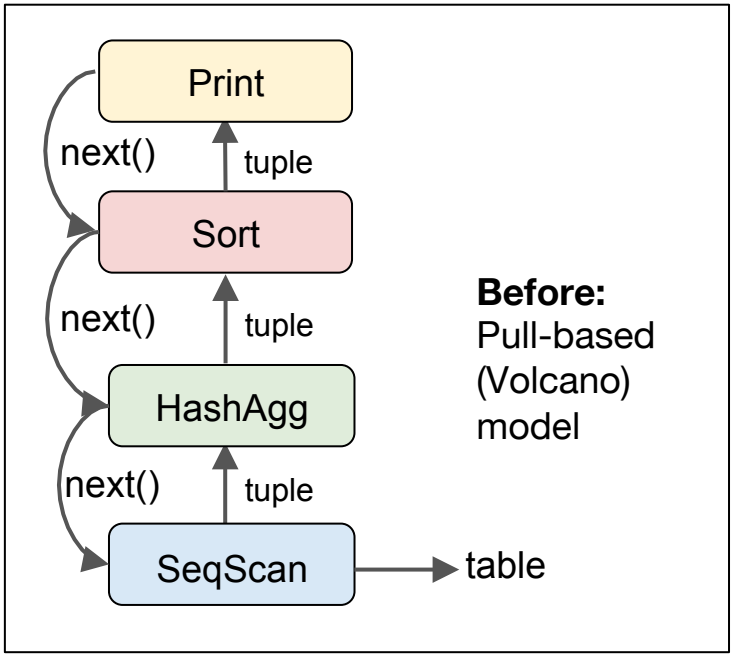
ISP RAS

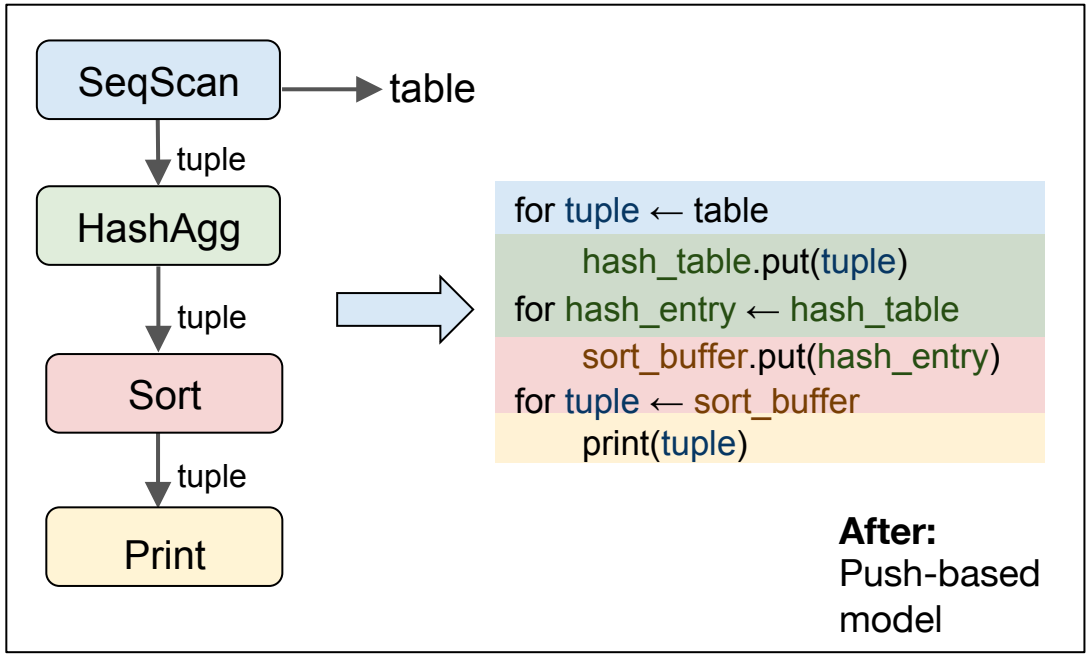


Changing Execution Model

ISP RAS

```
select <columns>                    HashAgg
from <table> where <condition>      SeqScan  Filter
group by <column>
order by <column>;                  Sort
```





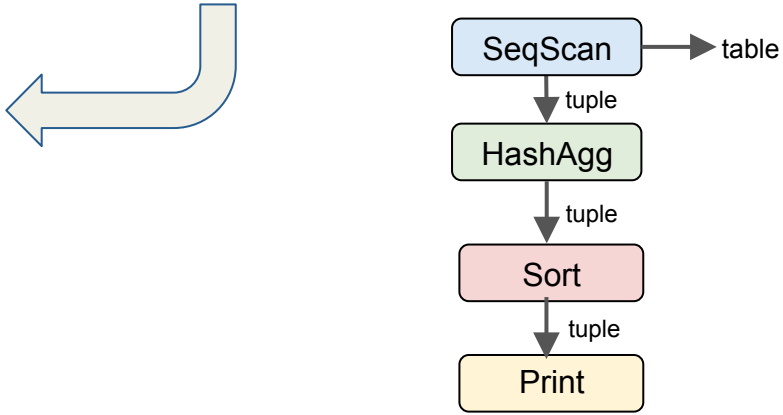
Plan Execution: push-based model

LLVM IR

```
llvm.sort.consume(tuple) {
    sort_buffer.put(tuple)
}
llvm.sort.finalize() {
    for tuple ← sort_buffer
        print(tuple)
}
```

LLVM C API

```
llvm.sort.consume = Sort.gen_consume()
llvm.sort.finalize = Sort.gen_finalize(print, null)
```



Plan Execution: push-based model

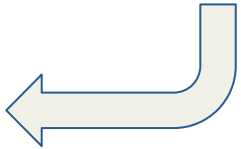
ISP RAS

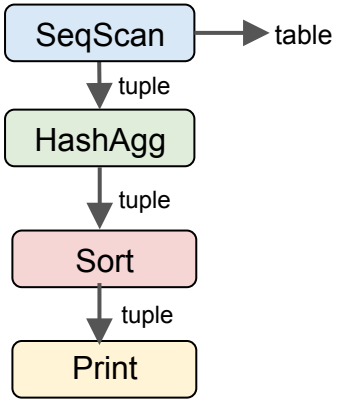
LLVM IR

```
llvm.agg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.agg.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)
}
```

LLVM C API

```
llvm.sort.consume = Sort.gen_consume()
llvm.sort.finalize = Sort.gen_finalize(print, null)

  llvm.agg.consume = HashAgg.gen_consume()
  llvm.agg.finalize = HashAgg.gen_finalize(llvm.sort.consume, llvm.sort.finalize)
```





Plan Execution: push-based model

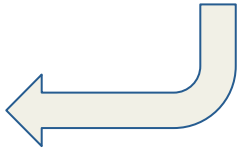
ISP RAS

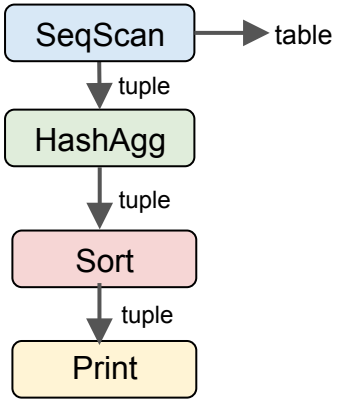
LLVM IR

```
llvm.scan() {
    for tuple ← table
        llvm.agg.consume(tuple)
    llvm.agg.finalize()
}
llvm.agg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.agg.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)
}
```

LLVM C API

```
llvm.sort.consume = Sort.gen_consume()
llvm.sort.finalize = Sort.gen_finalize(print, null)

  llvm.agg.consume = HashAgg.gen_consume()
  llvm.agg.finalize = HashAgg.gen_finalize(llvm.sort.consume, llvm.sort.finalize)

    llvm.scan = SeqScan(llvm.agg.consume, llvm.agg.finalize)
```



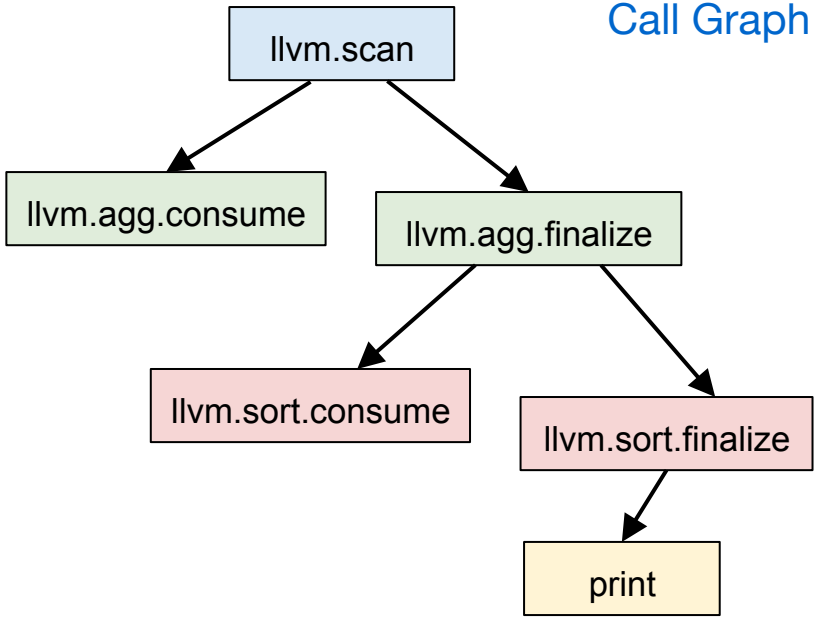


Plan Execution: push-based model

LLVM IR

```
llvm.scan() {
    for tuple ← table
        llvm.agg.consume(tuple)
    llvm.agg.finalize()
}
llvm.agg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.agg.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)
}
```

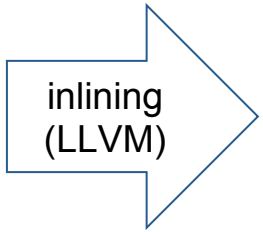
Call Graph



Plan Execution: push-based model

LLVM IR

```
llvm.scan() {
    for tuple ← table
        llvm.agg.consume(tuple)
    llvm.agg.finalize()
}
llvm.agg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.agg.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)
}
```



LLVM IR

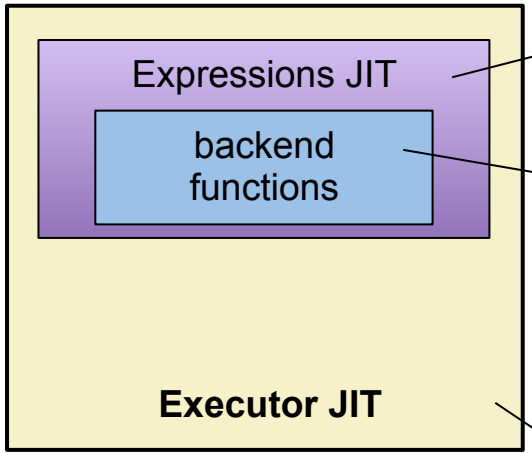
```
main() {
    for tuple ← table
        hash_table.put(tuple)
    for hash_entry ← hash_table
        sort_buffer.put(hash_entry)
    for tuple ← sort_buffer
        print(tuple)
}
```

- No indirect calls
- No need to store internal state

JIT Compilation at Different Levels

ISP RAS

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
```

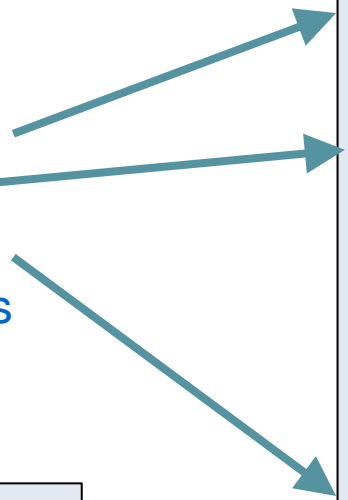
3%

20%

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

JIT-compiling attribute access

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

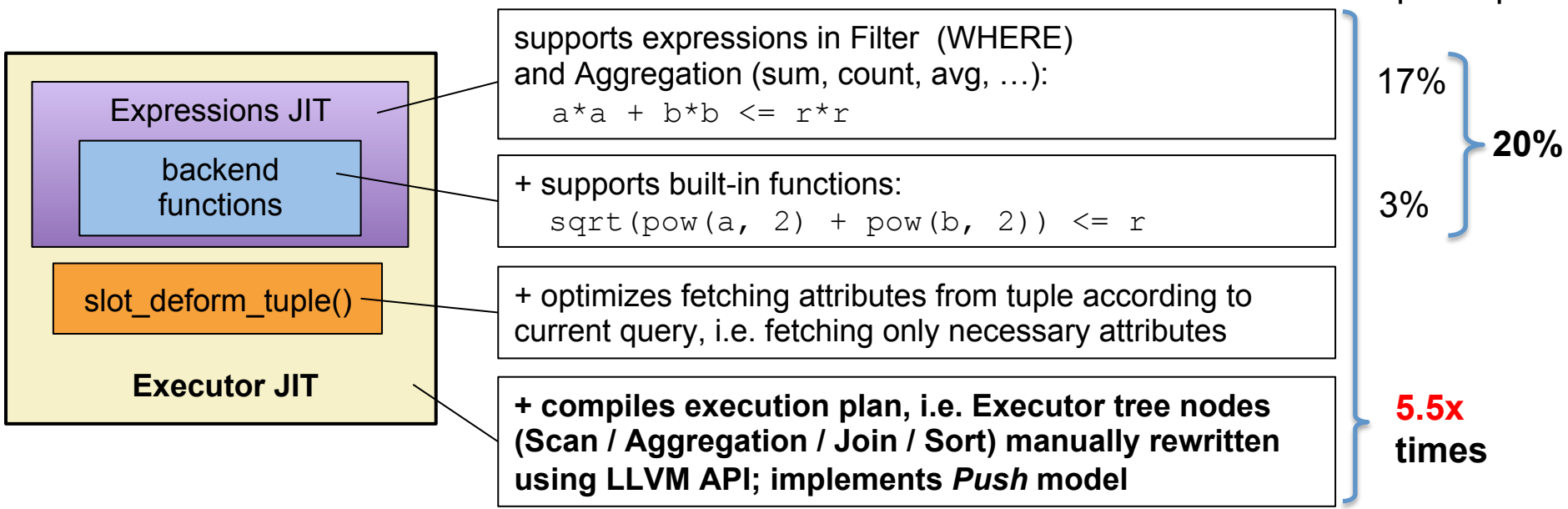


```
for (attnum = 0; attnum < natts; attnum++) {
    Form_pg_attribute thisatt = att[attnum];

    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



- Implemented as an extension for PostgreSQL 9.6.3
- 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

ISP RAS

- Measured on PostgreSQL 9.6.3, 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	110,03	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	58,23	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	1,57	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**	**1,90**	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

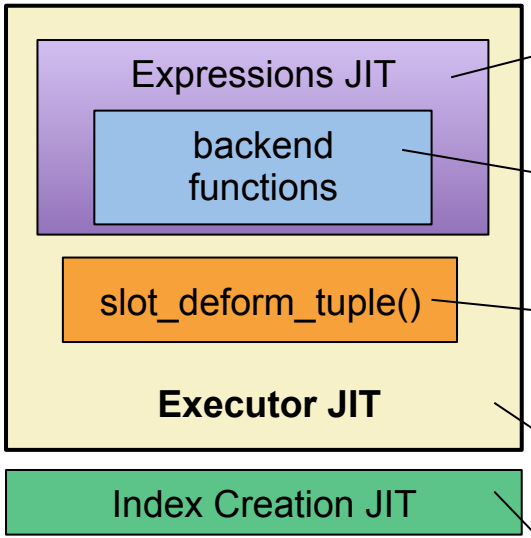
JIT-compiling Index Creation

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

JIT Compilation Everywhere

ISP RAS



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
```

+ 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

+ Compiling creation of *btree* index

TPC-H Q1 speedup ~

17%

3%

20%

5.5x times

5-19%

Results for Executor JIT

ISP RAS

- Measured on PostgreSQL 9.6.3, 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	110,03	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	58,23	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**	**1,57**	**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**	**1,90**	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

ISP RAS

Modified code generation to add needed indirection for simple query:

```
prepare qz as select * from lineitem where l_quantity < 1;
execute qz;
```

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 finished our work 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
- We have to backup LLVM-generated “.data” section and restore it before each next execution (initialize global vars for native code)
- For simple query (next slide) CachedPlan uses 3384 bytes in memory, and patchpointed-code is 5112 bytes (original code 1704 bytes).
- 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)
- It is 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%)

OLTP-query example

ISP RAS

```
Pgbench TPC-B-like database, scale=10 (DB size ~500mb),
1000 transactions
\set bid random(1, :scale)
\set a1 random(-25000, 25000)
\set a2 random(:a1, 25001)
SELECT aid FROM pgbench_accounts WHERE
    bid = :bid and :a1 <= abalance and abalance <= :a2 LIMIT 20;
```

	Simple query, vanilla PG	Simple query, JIT extension	Prepared query, vanilla PG	Prepared query, JIT extension
TPS	21.8	7.8	20.7	43
latency avg, ms	45	127	48	23
latency stddev	27	12	29	18

~2 times speedup on average

Prepared TPC-H queries

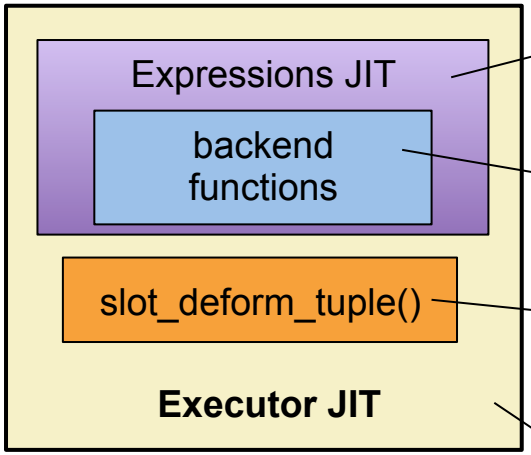
ISP RAS

- Measured on PostgreSQL 9.6, extension with LLVM 4.0 MCJIT, Core i7 CPU
- Database: 1GB (RamDisk storage, folder size ~3.5GB), shared_buffers = 4GB

Query	Postgres	LLVM JIT extension			Patchpoint-JIT for PREPARED queries			Speedup, times	JIT exec slowdown	Patchpoint code
		Compilation	Execution	Total	Compilation	Execution*	Total			
Q01	**3,05**	0,72	0,61	1,33	0,66	**0,75**	1,41	**4,08**	22%	30%
Q03	**1,21**	0,87	0,45	1,32	1,01	**0,53**	1,54	**2,28**	16%	40%
Q09	**1,13**	1,43	0,86	2,29	1,65	**0,86**	2,51	**1,33**	0%	41%
Q13	**1,31**	0,71	0,80	1,51	0,77	**0,84**	1,61	**1,55**	5%	24%
Q15	**1,70**	0,87	0,50	1,37	1,00	**0,69**	1,69	**2,48**	39%	36%
Q18	**1,81**	1,02	0,72	1,74	1,20	**0,78**	1,98	**2,34**	7%	50%
Q19	**0,06**	0,90	0,03	0,93	0,87	**0,03**	0,90	**1,88**	0%	27%
Q21	**0,78**	0,87	0,70	1,57	0,90	**0,70**	1,60	**1,11**	0%	27%

JIT Compilation Everywhere

ISP RAS



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
```

+ 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

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

+ Compiling creation of *btree* index

TPC-H Q1 speedup ~

17%

3%

20%

5.5x times

+ eliminates compilation overhead

5-19%

Results (1)

ISP RAS

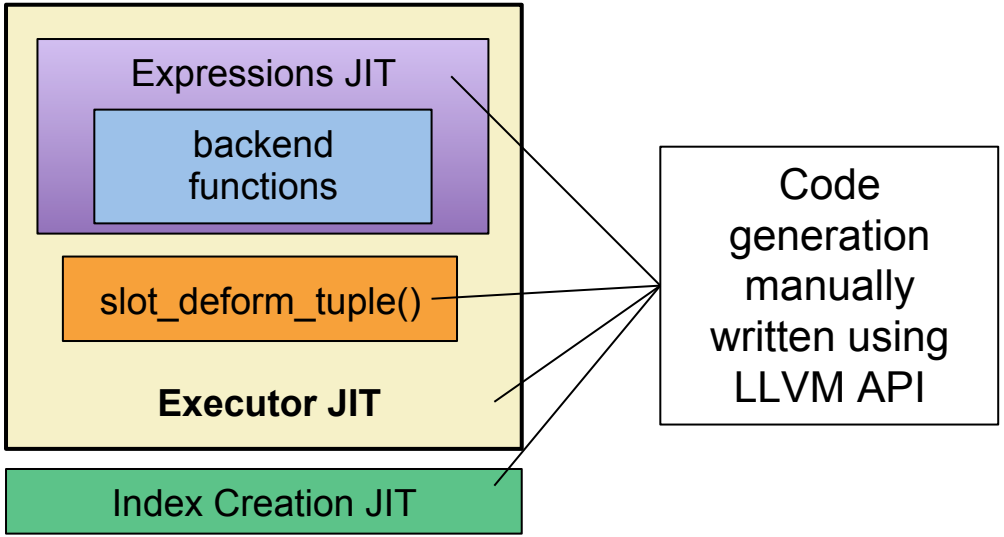
- Expression JIT
  - Open source: github.com/ispras/postgres
  - Speedup up to **20%** on TPC-H
- PostgreSQL Extension JIT (still developing)
  - Speedup up to **5.5 times** on TPC-H
- Caching JITted code for PREPARED statements
  - Eliminate overhead on JIT compilation, useful for OLTP
- Index creation JIT
  - Up to **19%** speedup

Agenda

ISP RAS

- Expression JIT
- Full Executor JIT
- Caching JITted code for PREPARED statements
- Index creation JIT
- **Experimental**
  - Run time Executor code specialization for a given query
  - Switching original PosgreSQL Executor from pull to push model

Automatic JIT generation?

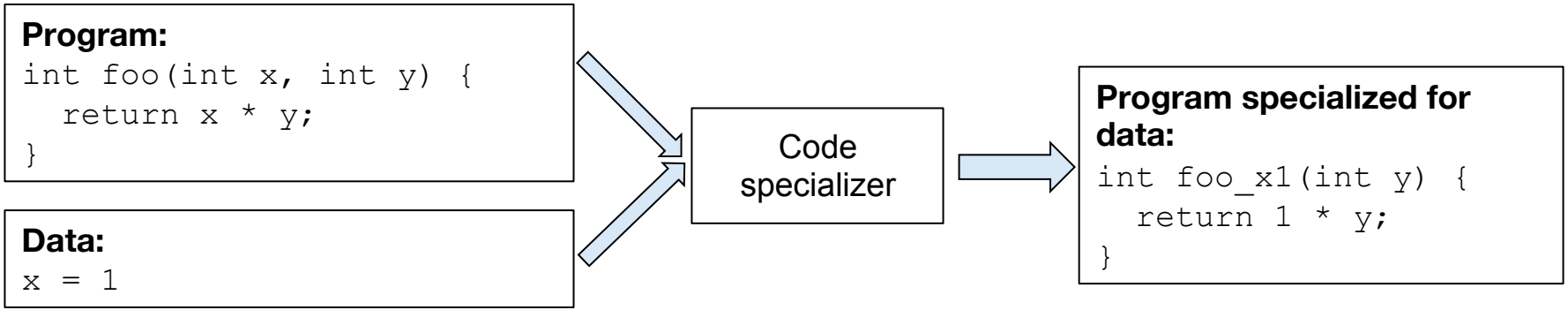


Challenges:

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

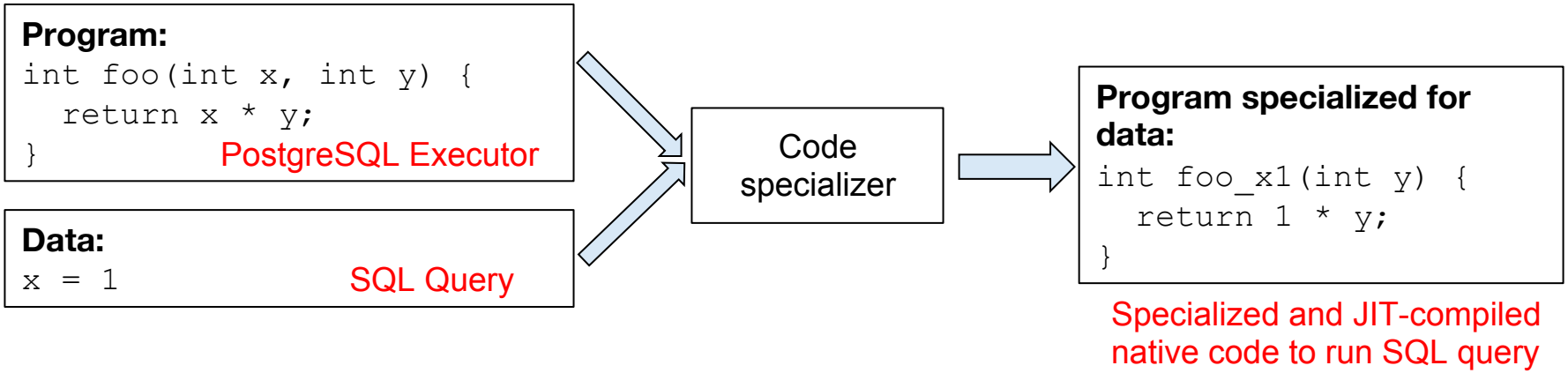
Automatic code specialization could be the answer

Run time code specialization



- Optimizations that can be performed:
  - Constant propagation / folding
  - Simplify control flow, unroll loops, eliminate conditional branches
  - Statically evaluate expressions
  - …

Run time code specialization

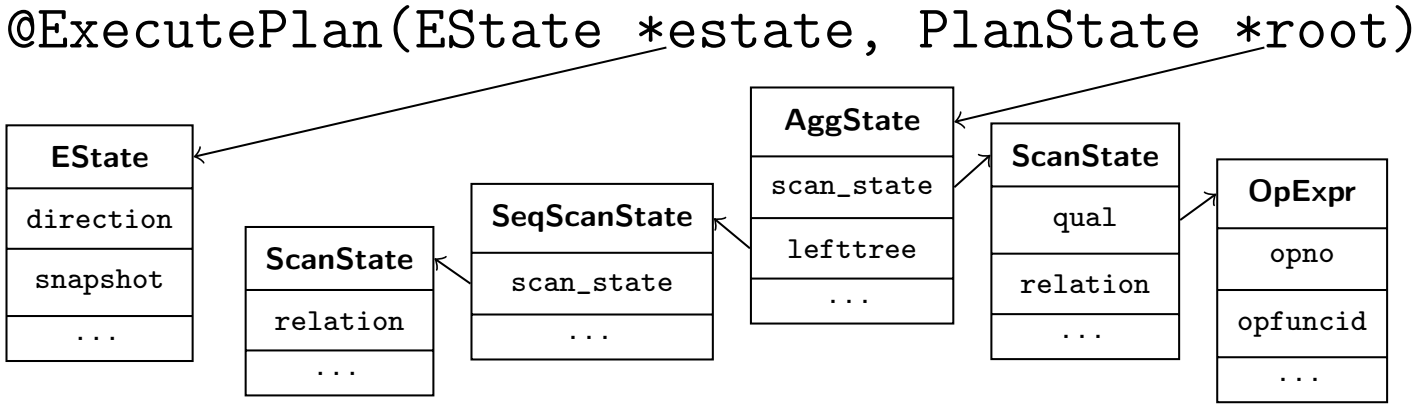


- For a given SQL query in Postgres:
  - static data (Plan, PlanState, EState) — depend only from SQL query itself, and are invariant during query execution
  - dynamic data (HeapTuple, ItemPointer) — depend both from SQL query and data

Constant Propagation for Specialized Code

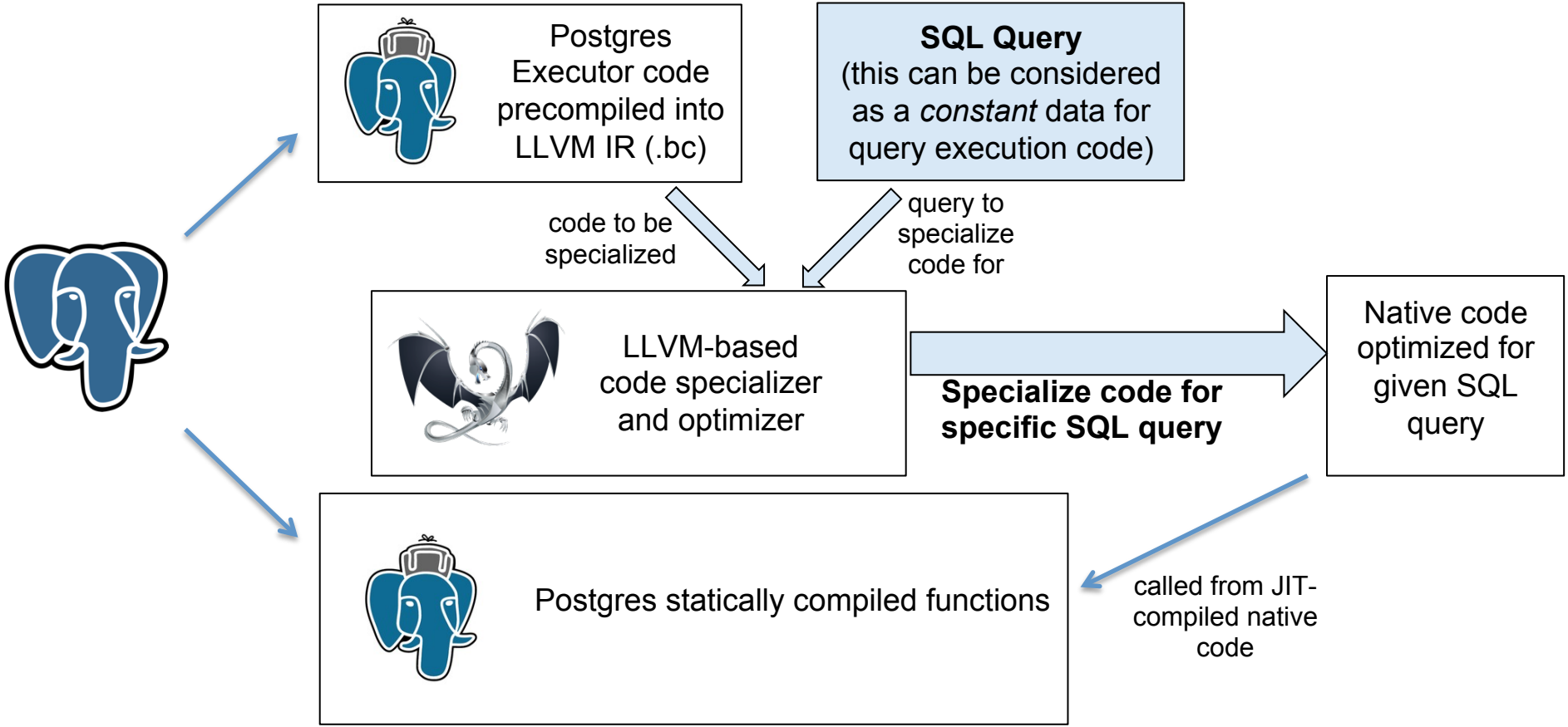
- Make constant propagation work for invariant values on the heap
  - We know what data is invariant for the given SQL query
  - LLVM already has constant propagation / folding optimizations
  - We need to save addresses holding constant values to make LLVM use them in optimizations

Constant Propagation for Specialized Code



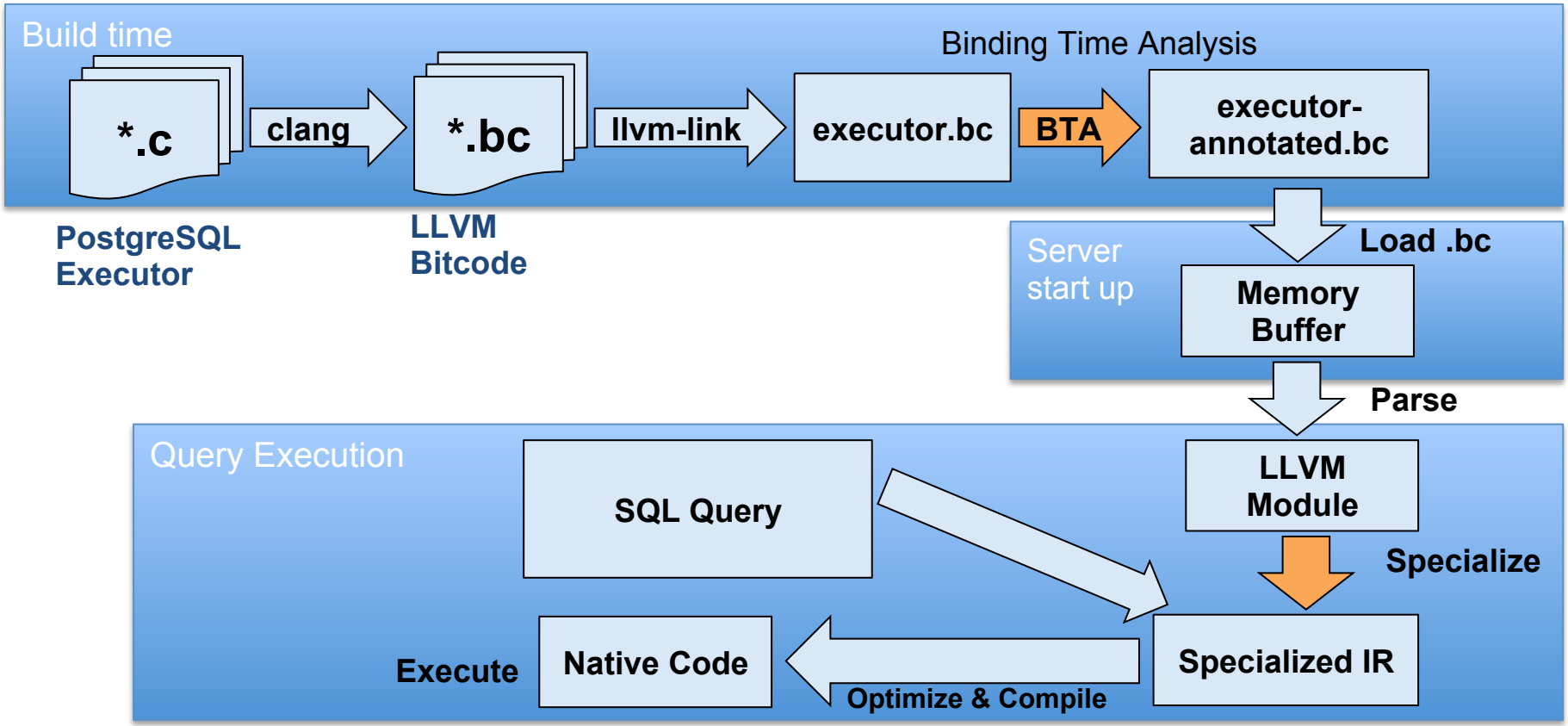
- Offline: manually mark up invariant fields in data structures (Plan, PlanState, Estate)
- JIT compilation at run time:
  - Traverse data structures graph for function arguments, store addresses with invariant data
  - Run LLVM optimizer to substitute memory loads from those addresses to values they hold at the time of JIT compilation

Run time code specialization



Specializing PostgreSQL Executor Code

ISP RAS



Run-time specialization

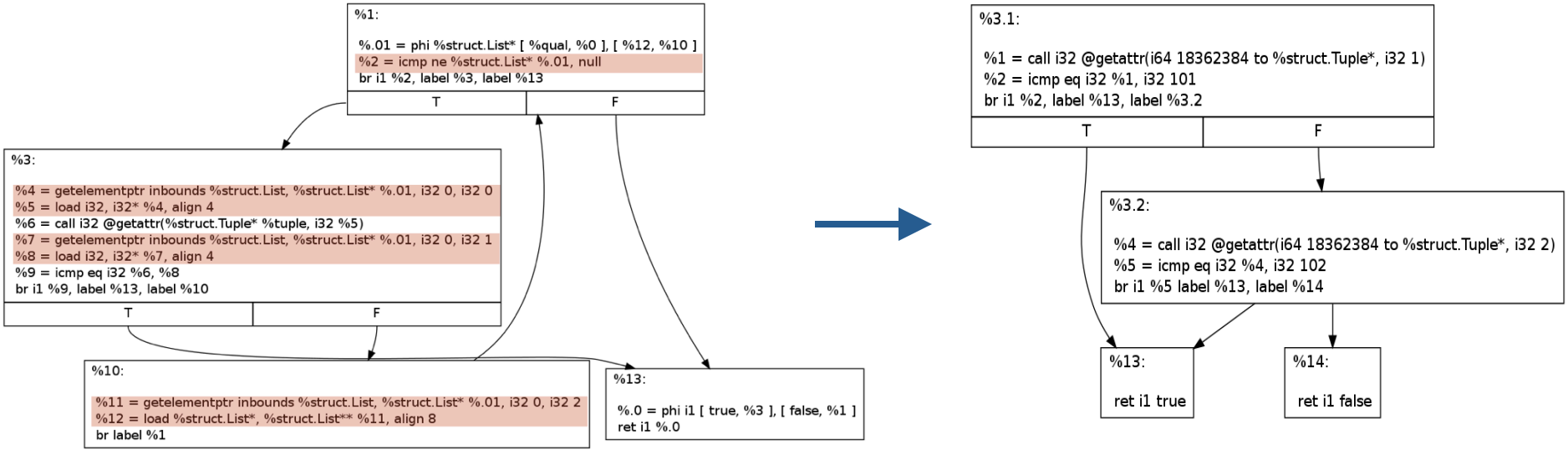
```
for (attnum = 0; attnum < natts; attnum++) {
    Form_pg_attribute thisatt = att[attnum];

    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);
}
```

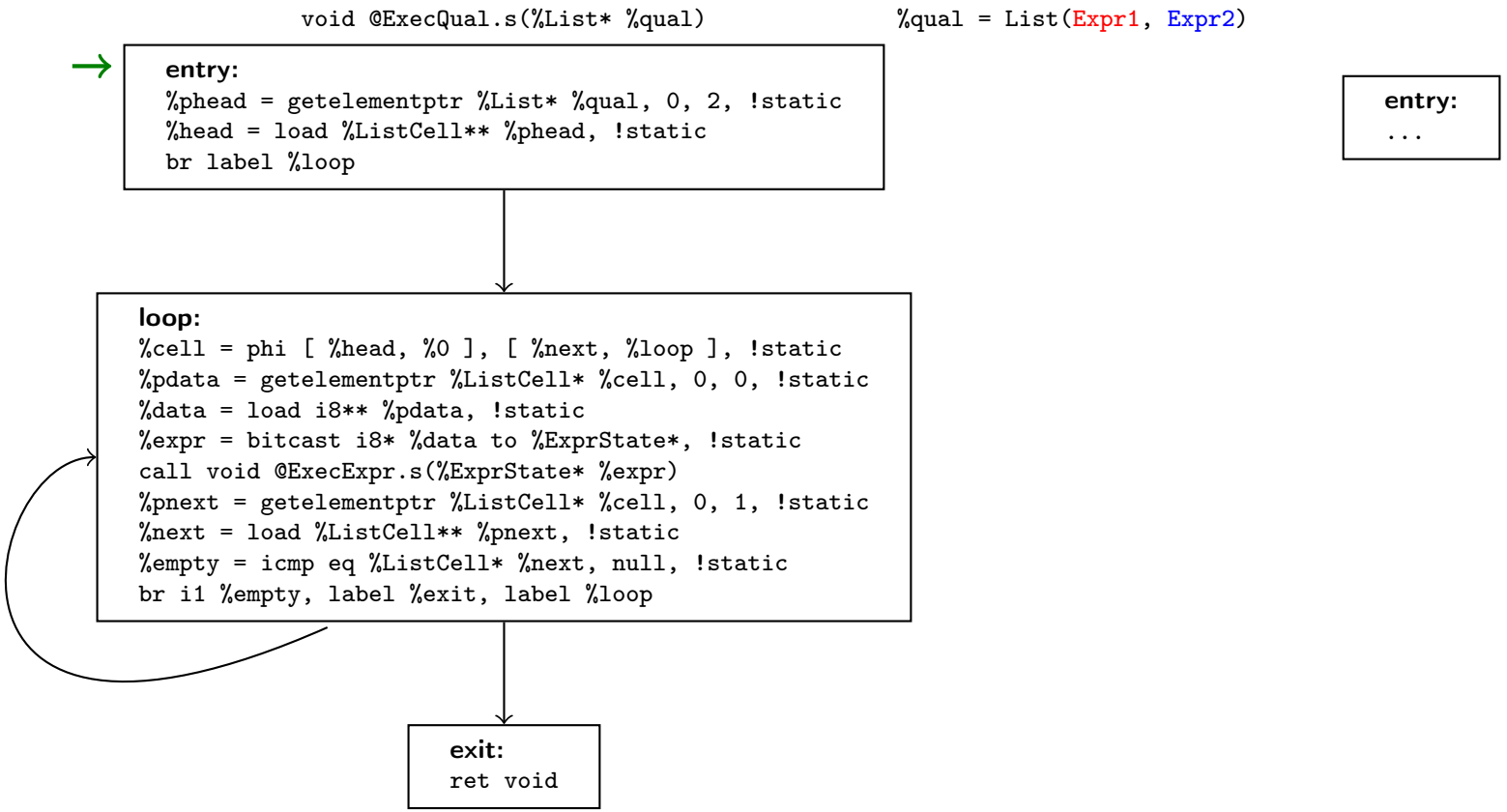
Data available at run-time specialization is marked in RED

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

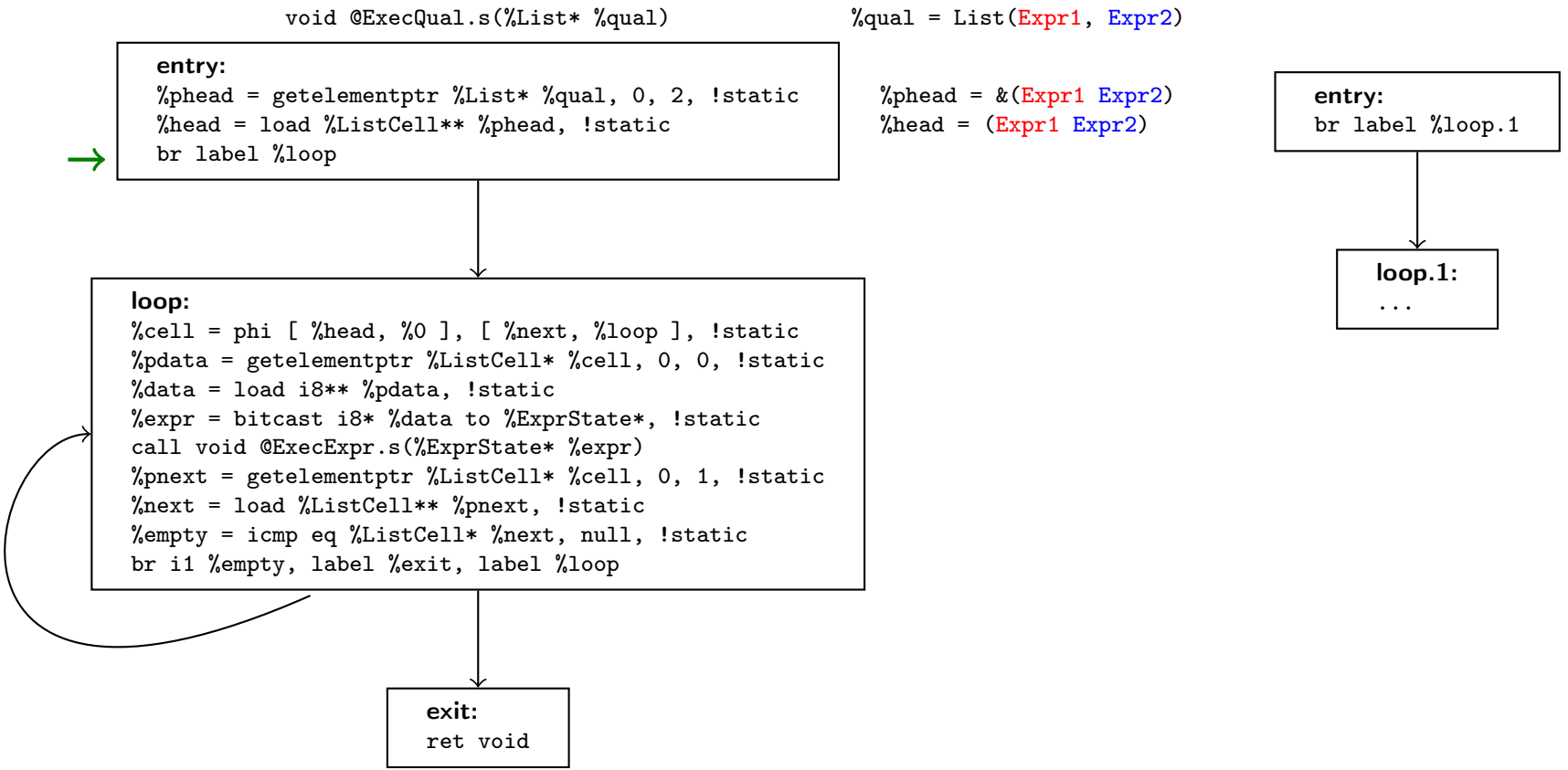
Run time code specialization



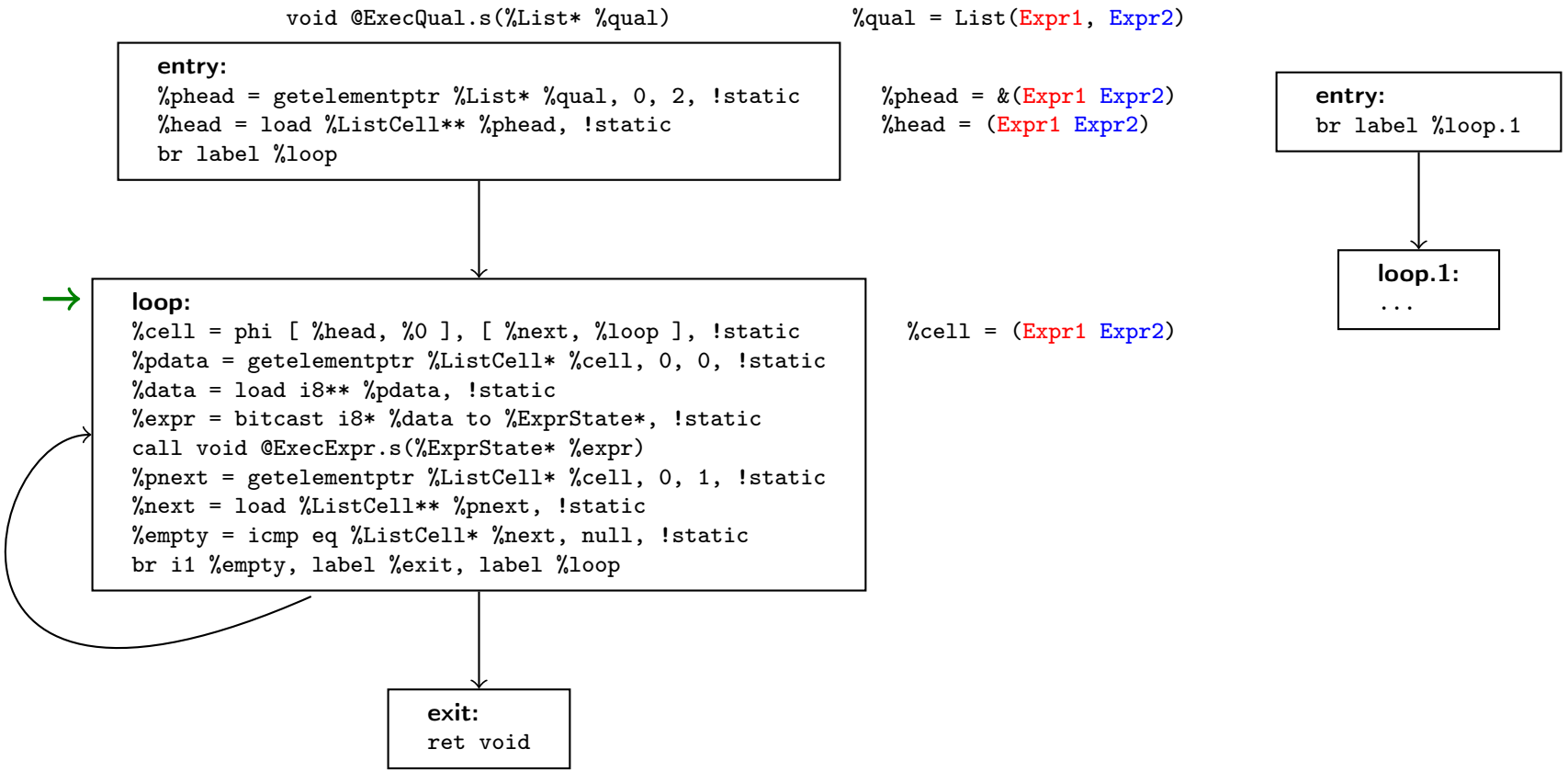
Specializing a function



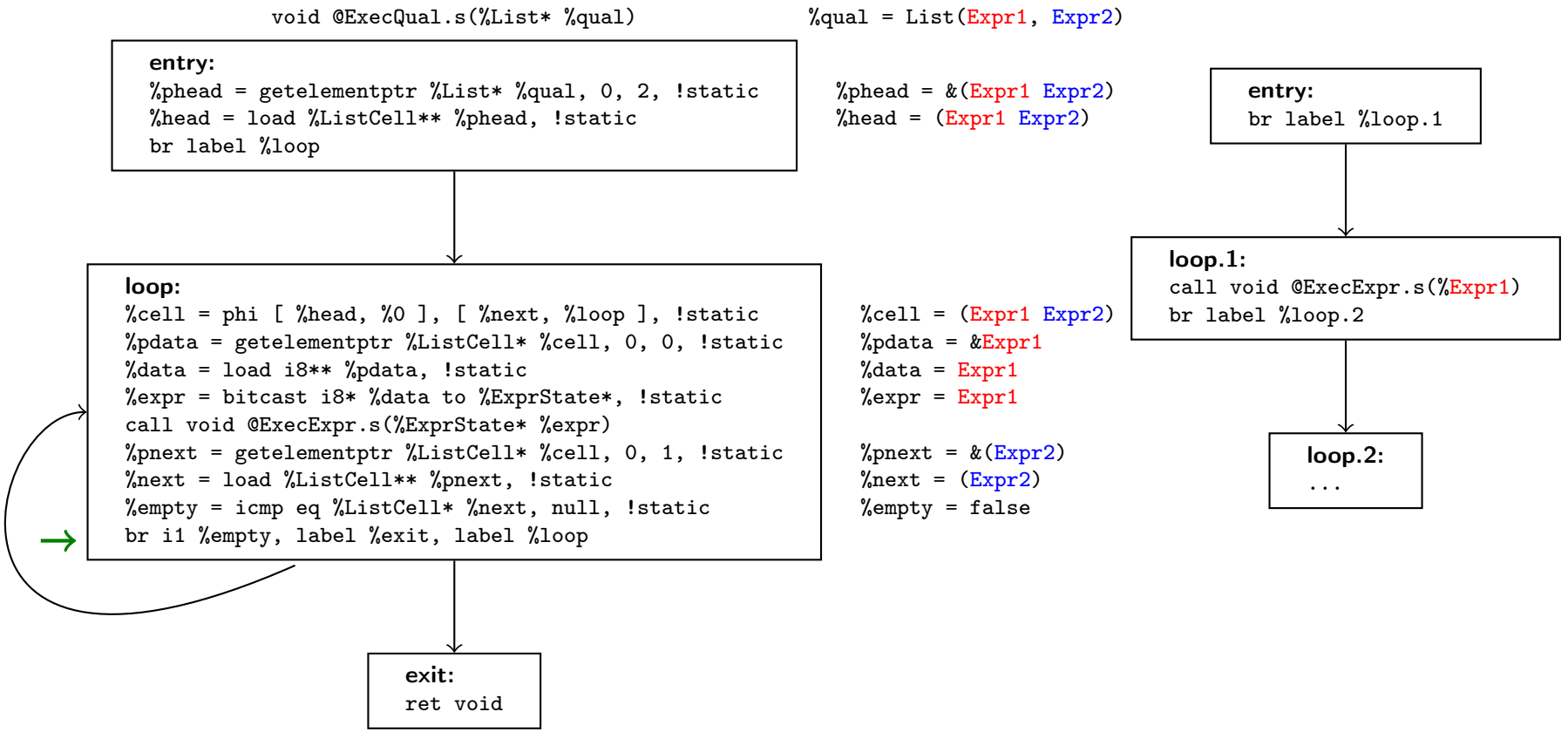
Specializing a function



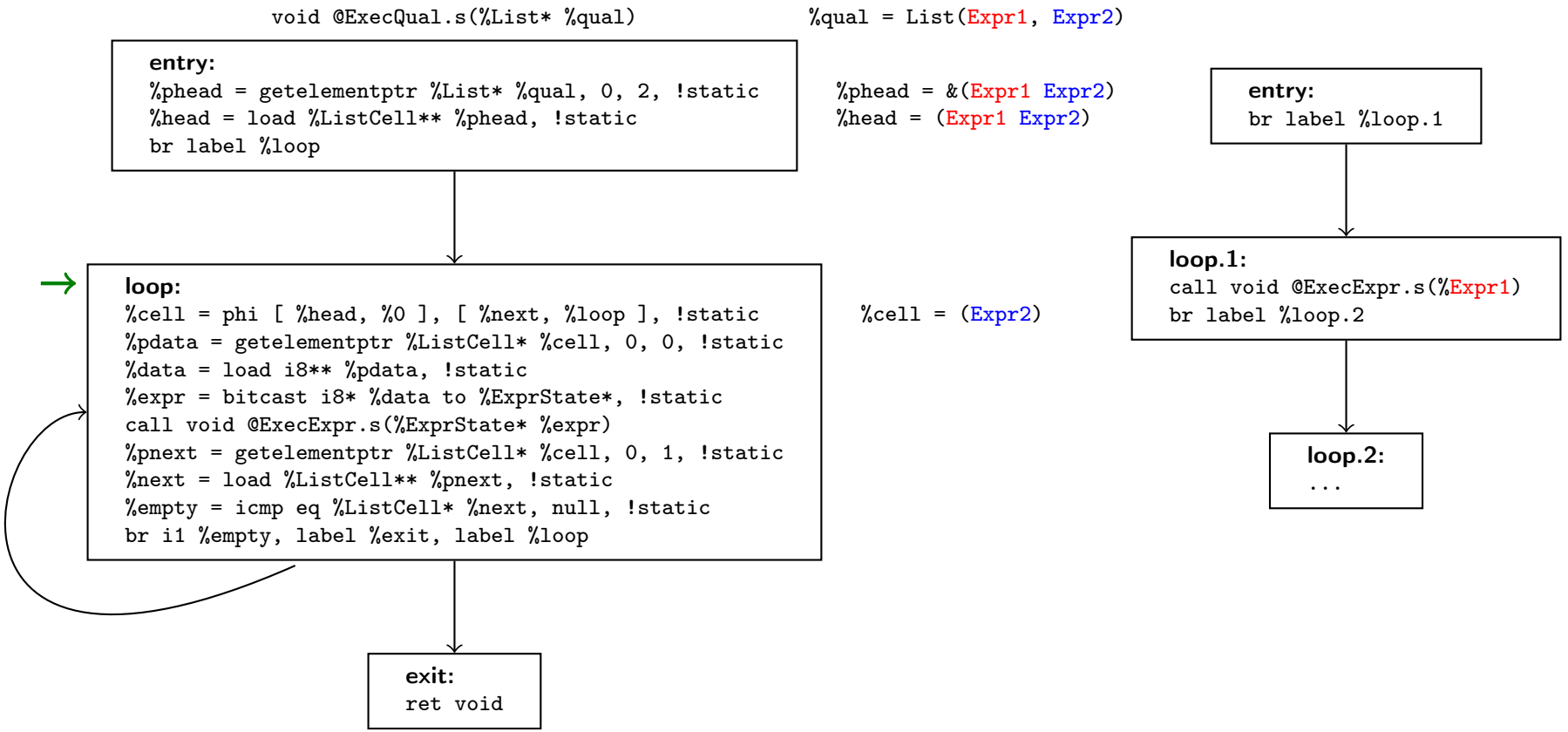
Specializing a function



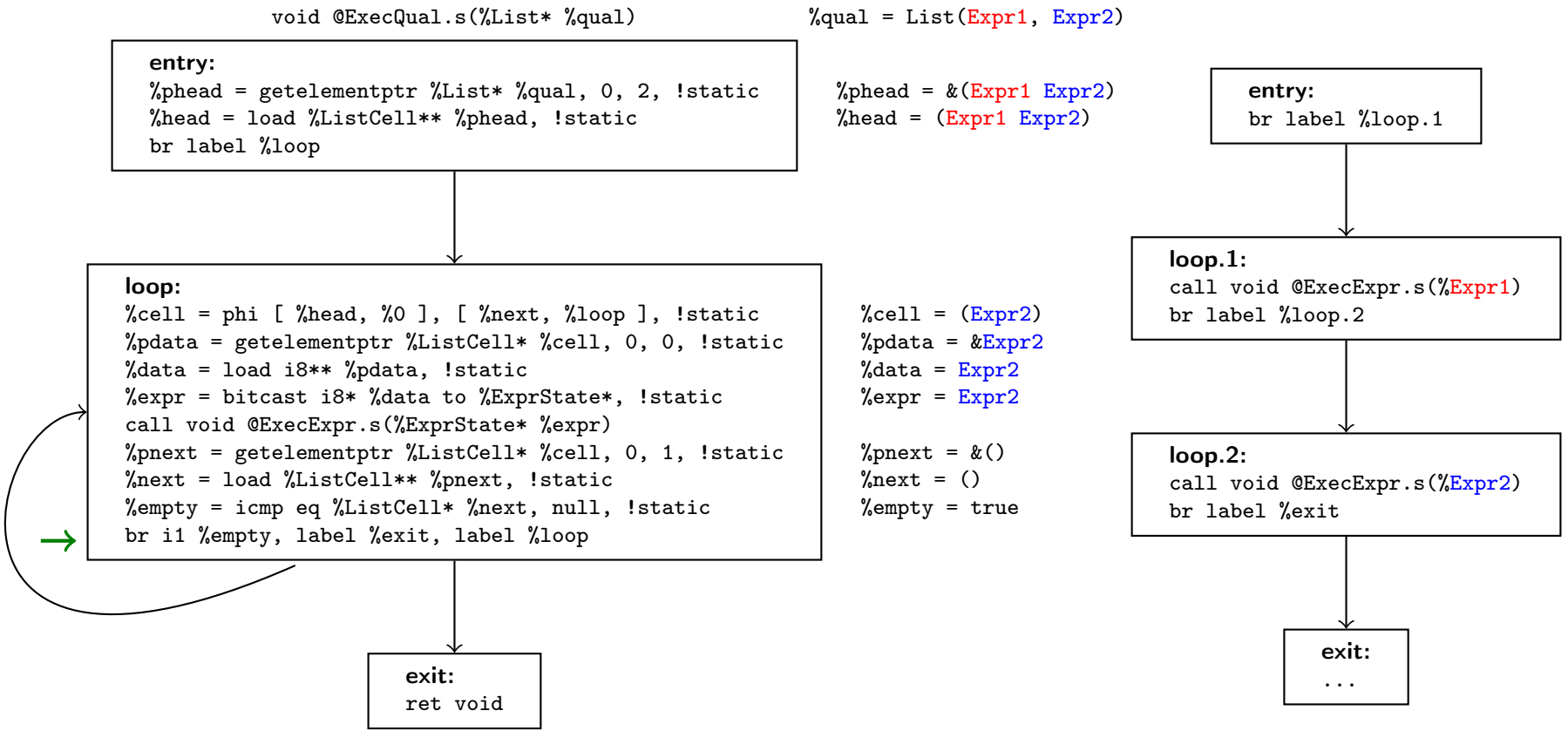
Specializing a function



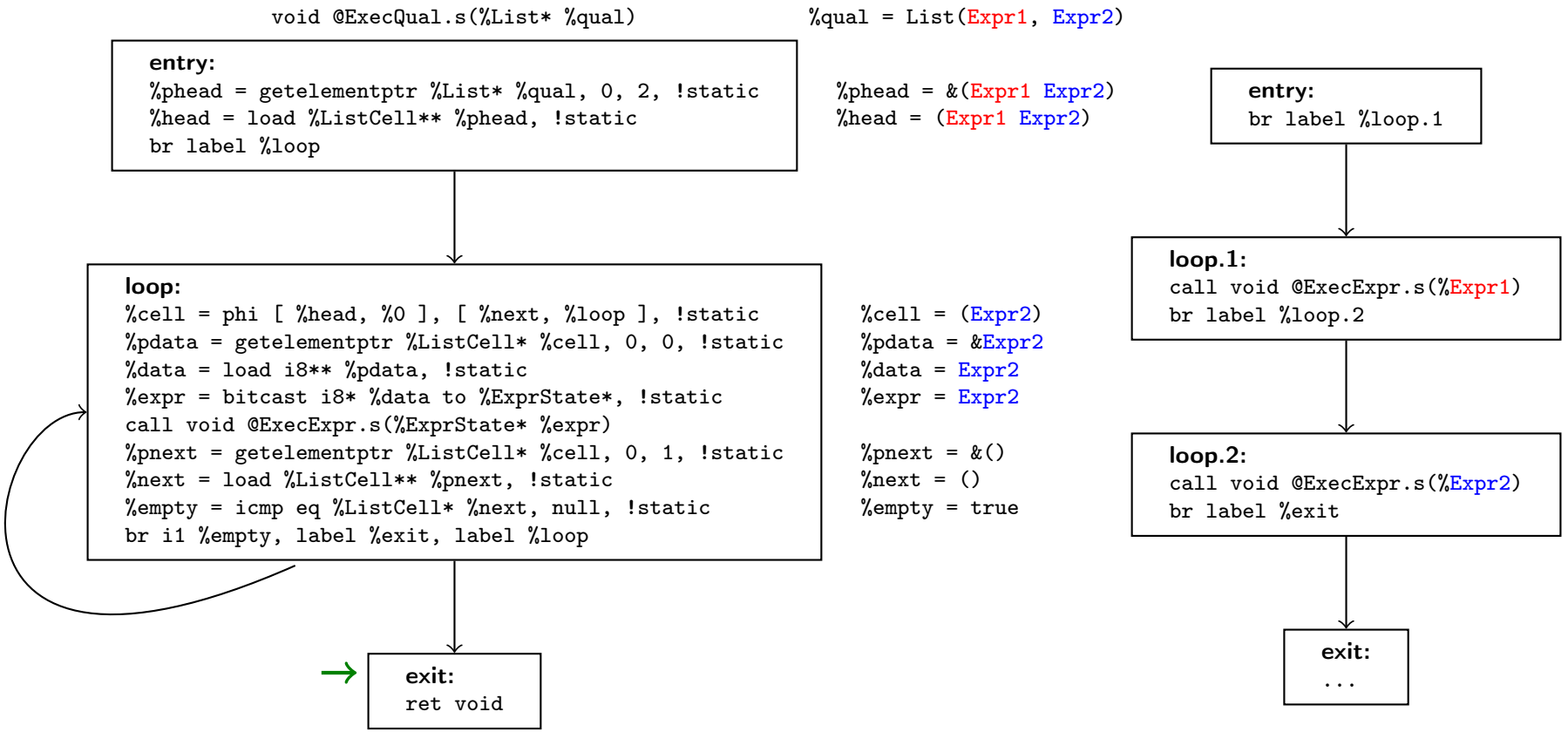
Specializing a function



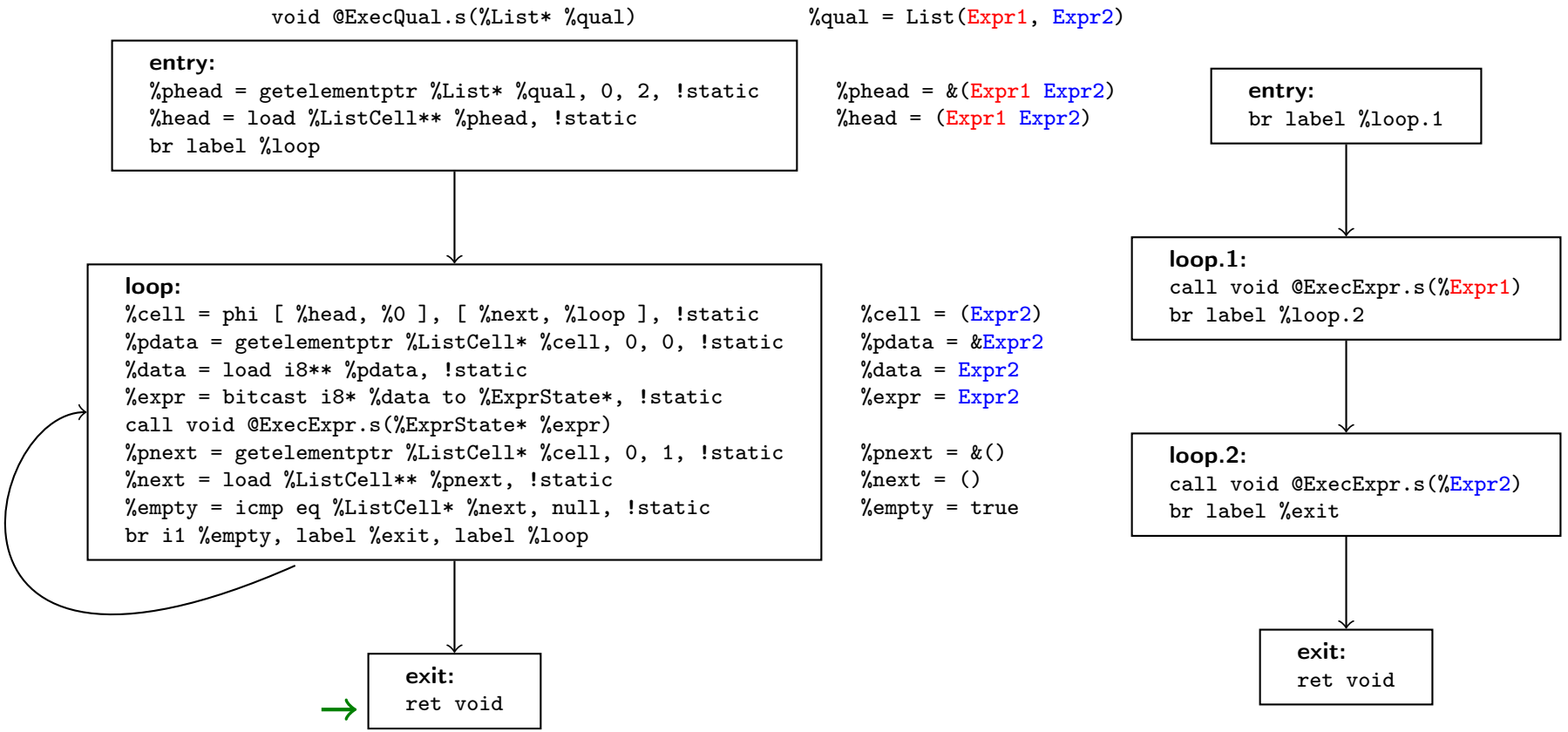
Specializing a function



Specializing a function



Specializing a function



Preliminary Results for Executor Specialization

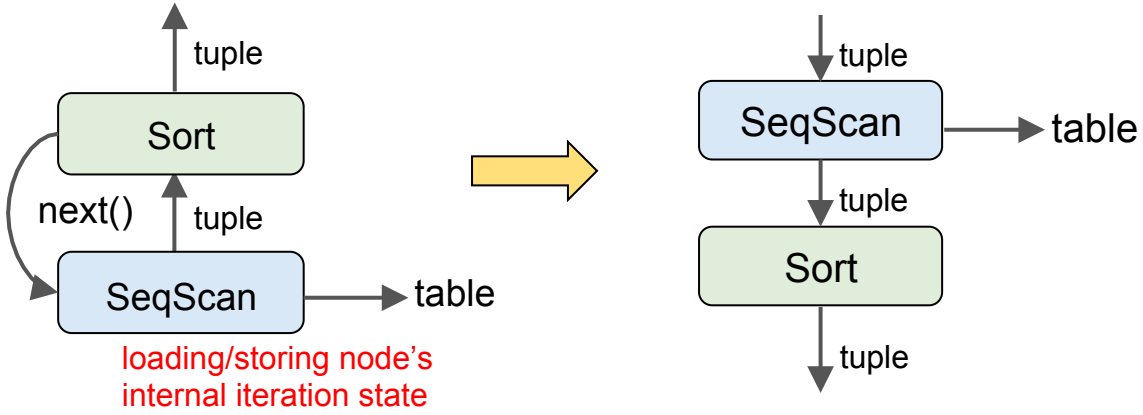
ISP RAS

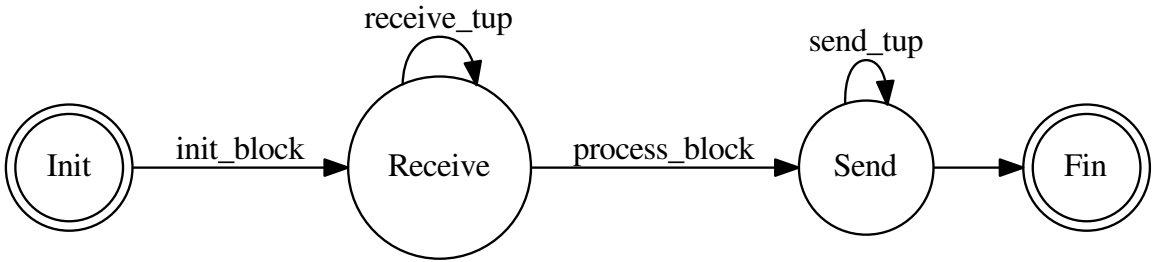
- Up to 28% speedup on synthetic test queries
- Storing/loading node state at leaf nodes hampers the performance

Switching to Push Model in Original PostgreSQL Executor

ISP RAS

```
select * from horns order by horns_weight;
```





Benefits for Push Model

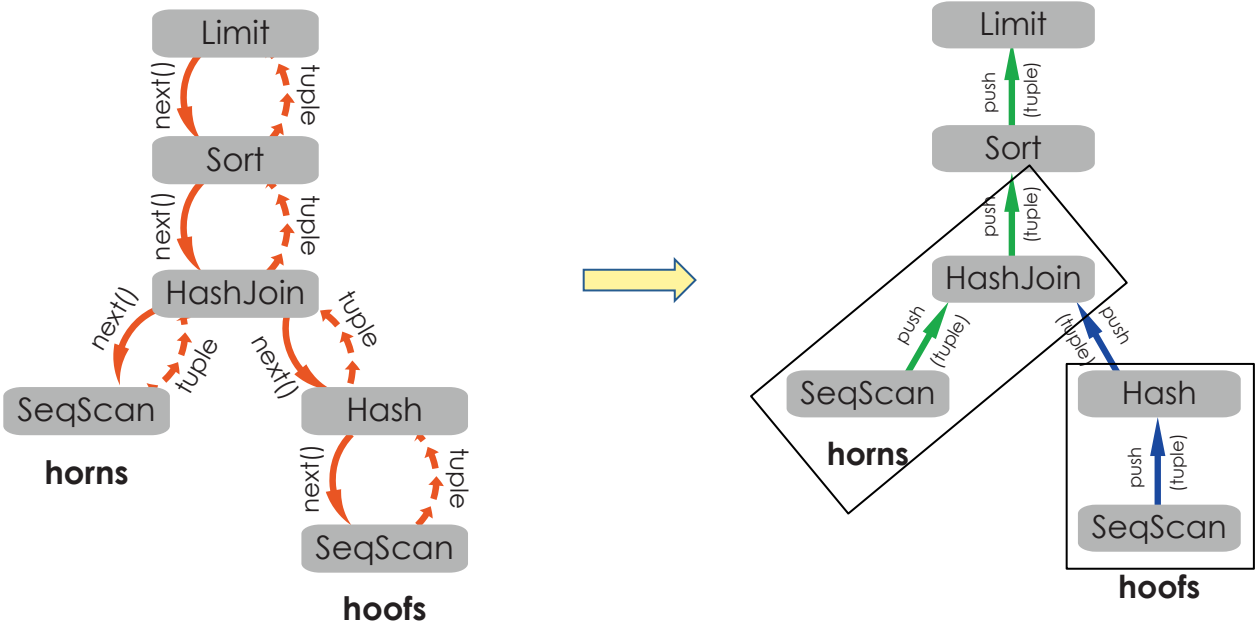
ISP RAS

- More natural way to iterate leaf Executor nodes
- Can keep iteration variables on stack, don’t have to load/store node iteration state on heap
- Better code and data locality

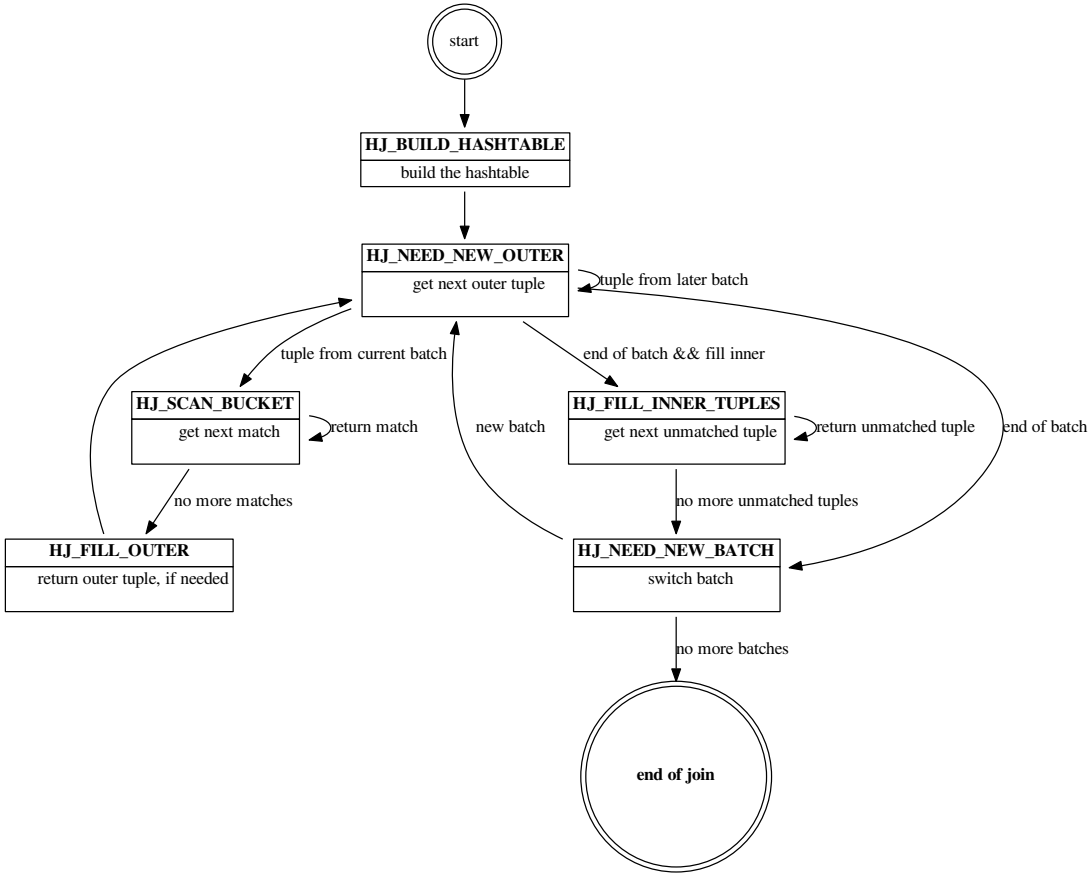
Switching to Push Model in Original PostgreSQL Executor

ISP RAS

```
select * from horns join hoofs on horns.sheep_id =
hoofs.sheep_id order by horns.weight + hoofs.weight limit 4;
```



HashJoin Automaton



Switching to Push Model in Original PostgreSQL Executor

ISP RAS

Query #	Pull, sec	Push, sec	Speedup, %
1	72.97	67.49	8.12
3	35.04	33.47	4.69
4	48.07	45.15	6.47
5	31.55	27.79	13.53
6	19.4	18.37	5.61
10	31.55	30.15	4.64
12	38.74	36.94	4.87
13	38.63	36.67	5.34
14	20.68	19.45	6.32
19	30.57	28.45	7.45
21	125.92	120.49	4.51

TPC-H queries time, scale=25

- Implemented SeqScan, Hash, HashJoin, Limit, Agg and Sort
- ~5-10% speedup on TPC-H queries
- Profiling with `perf` shows that speedup comes mainly from SeqScan optimization

Conclusions (for Switching Original Executor for Push Model)

ISP RAS

- Noticeably reduces overhead for leaf Executor tree nodes (like SeqScan)
- 5-10% speedup on TPC-H queries
- Implemented prototype is published in mailing list: https://www.postgresql.org/message-id/87r31pxouh.fsf%40ispras.ru
- Dynamic specializer works better with push model, but needs additional tuning

Conclusions

ISP RAS

- 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 JITted code for PREPARED statements
  - Eliminate overhead on JIT compilation, useful for OLTP
- Index creation JIT
  - Up to **19%** speedup
- Experimental:
  - Run time code specialization: up to 28% speedup
  - Switching original PostgreSQL executor from pull to push model

Conclusions

ISP RAS

- Calling for Collaboration!
  - Will be happy to get your feedback and discuss further development of JIT compilation in PostgreSQL
  - Looking for partners in the industry to tune JIT compiler for real world applications and test on different workloads

Questions, comments, feedback:

dm@ispras.ru

Thank you!