Just In Time Compilation in PostgreSQL 11 and onward

Andres Freund

PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec

anarazel.de/talks/2018-09-07-pgopen-jit/jit.pdf

Motivation

TPC-H Q01

```
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 '74 days'
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

TPC-H Q01

```
Sort  (cost=4313208.98..4313209.00 rows=6 width=68)
      (actual time=33983.596..33983.596 rows=4 loops=1)
  Sort Key: l_returnflag, l_linestatus
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=4 read=1186601
  I/O Timings: read=6124.546
  ->  HashAggregate  (cost=4313208.80..4313208.91 rows=6 width=68)
                     (actual time=33982.837..33982.839 rows=4 loops=1)
        Group Key: l_returnflag, l_linestatus
        Buffers: shared hit=1 read=1186601
        I/O Timings: read=6124.546
        ->  Seq Scan on lineitem  (cost=0.00..1936377.20 rows=59420790)
                                  (..time=13841.766 rows=59414087 loops=1)
              Filter: (l_shipdate <= '1998-09-18 00:00:00'::timestamp)
              Rows Removed by Filter: 571965
              Buffers: shared hit=1 read=1186601
              I/O Timings: read=6124.546
Planning Time: 29.888 ms
Execution Time: 33984.546 ms
```

TPC-H Q01

```
Sort  (cost=4313208.98..4313209.00 rows=6 width=68)
      (actual time=26509.669..26509.670 rows=4 loops=1)
  Sort Key: l_returnflag, l_linestatus
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=1186602
  ->  HashAggregate  (cost=4313208.80..4313208.91 rows=6 width=68)
                     (actual time=26509.622..26509.625 rows=4 loops=1)
        Group Key: l_returnflag, l_linestatus
        Buffers: shared hit=1186602
        ->  Seq Scan on lineitem  (cost=0.00..1936377.20 rows=59420790 width=36)
                                  (time=0.016..8132.990 rows=59414087 loops=1)
              Filter: (l_shipdate <= '1998-09-18 00:00:00'::timestamp)
              Rows Removed by Filter: 571965
              Buffers: shared hit=1186602
Planning Time: 5.161 ms
Execution Time: 26509.857 ms
```

TPC-H Q01 Profile

```
Samples: 87K of event 'cycles:ppp', cnt (approx.): 71706618234
  Overhead  Command   Shared Object          Symbol
-   35.96%  postgres  postgres               [.] ExecInterpExpr
      + 72.86% ExecAgg
      - 18.33% tuplehash_insert
           LookupTupleHashEntry
           ExecAgg
           ExecSort
      + 8.81% ExecScan
-   10.79%  postgres  postgres               [.] slot_deform_tuple
        slot_getsomeattrs
      - ExecInterpExpr
         + 77.31% ExecScan
         + 22.69% tuplehash_insert
+    4.96%  postgres  postgres               [.] tuplehash_insert
+    4.53%  postgres  postgres               [.] float8_accum
+    3.21%  postgres  postgres               [.] float8pl
+    2.61%  postgres  postgres               [.] bpchareq
+    2.40%  postgres  postgres               [.] hashbpchar
```

Solutions

- Micro (and not so micro) Optimizations
  - various pieces in v10, v11 and earlier releases, significant speedups
  - further possibilities, bottlenecks currently elsewhere
- Parallelism (close to linear scale in v10/11)
  - uses more resources
- Distributed Computation
  - uses more resources, out of core
- Columnar Store / Vectorized Execution
  - no mature, well integrated, postgres solutions exist
  - not commonly suitable for hybrid OLTP / OLAP workloads

What is “Just In Time” compilation

- “just-in-time (JIT) compilation, is a way of executing computer code that involves compilation during execution of a program – at run time – rather than prior to execution.” *

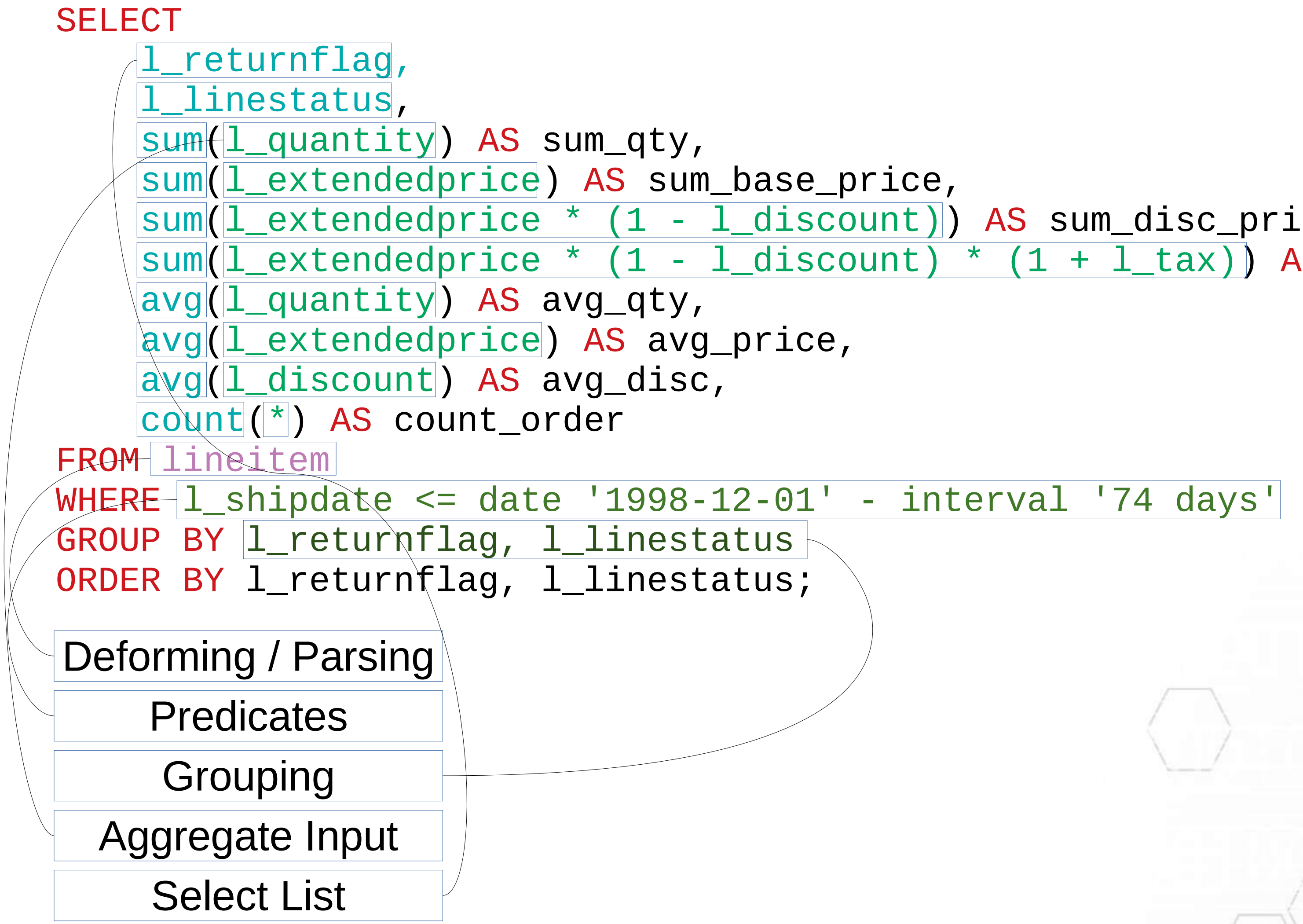
- Convert forms of “interpreted” code into native code
  - removes interpretation overhead
- Specialize code for specific constant arguments
  - removes entire “branches” of work
- Achieve speedups via:
  - reduced total number of executed instructions
  - reduced number of executed branches
  - reduced number of executed indirect jumps / calls
- Well known from browsers for javascripts, java VMs and the like

* https://en.wikipedia.org/wiki/Just-in-time_compilation

How does JIT compilation work in PostgreSQL

- Uses LLVM (llvm.org)
- Optional Feature (./configure --with-llvm)
- Doesn’t work on Windows at this point
- Packagers can install support separately (e.g postgresql11-llvmjit for yum.postgresql.org)
- `jit = on` && `SELECT pg_jit_available();`
- Extensible – other implementations / providers / extensions can replace (`jit_provider = ‘llvmjit’`)

Just-in-Time Compilation in v11



```
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 '74 days'
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

Deforming / Parsing

Predicates

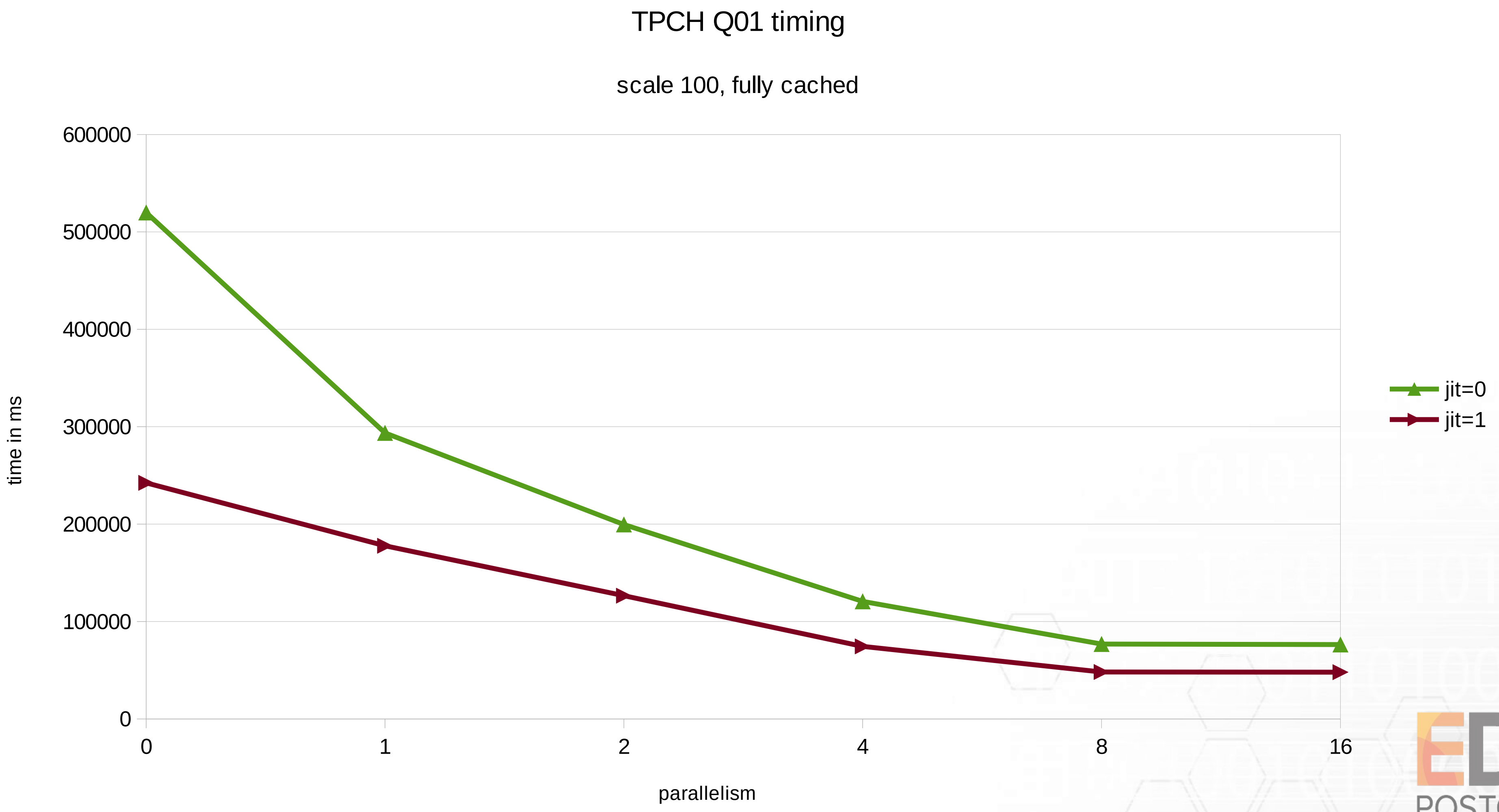
Grouping

Aggregate Input

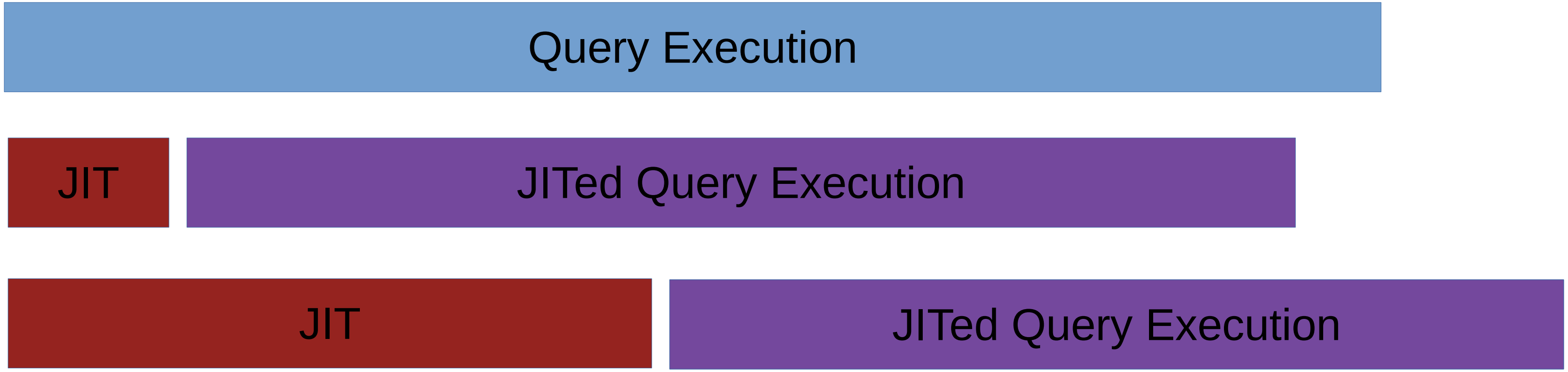
Select List

Parts of JIT Compilation in v11

- JIT compilation of expressions
  - removes interpretation of expressions, JIT compiles them
  - `jit = on/off`
- JIT compilation of tuple deforming
  - accellerates on-disk / buffer → in memory representation
  - `jit_tuple_deforming = on/off`
- Optimization of JIT compiled functions using LLVM
- Inlining of SQL level functions / operators
  - SQL defined operators have overhead that can be significant portion of time for simple functions
  - Available to extensions


TPCH Q01 timing
scale 100, fully cached
time in ms
600000
500000
400000
300000
200000
100000
0
0
1
2
4
8
16
parallelism
jit=0
jit=1

Planning JIT Compilation – Why not always



Planning JIT Compilation

- Naive!
- `jit = off` => no JIT compilation (default: on, but might change)
- Perform JIT if `query_cost > jit_above_cost`
  - `default: 100000`
- Optimize if `query_cost > jit_optimize_above_cost`
  - `default: 500000`
- Inline if `query_cost > jit_inline_above_cost`
  - `default: 500000`
- `-1` disables
- Whole query decision
- *NOT* a tracing JIT:
  - costing makes tracing somewhat superflous
  - tracing decreases overall gains

Planning JIT Compilation - Example

```
Sort  (cost=4313208.98..4313209.00 rows=6 width=68)
  Sort Key: l_returnflag, l_linestatus
  Sort Method: quicksort  Memory: 25kB
  ->  HashAggregate  (cost=4313208.80..4313208.91 rows=6 width=68)
        Group Key: l_returnflag, l_linestatus
        ->  Seq Scan on lineitem  (cost=0.00..1936377.20 rows=59420790 width=36)
              Filter: (l_shipdate <= '1998-09-18 00:00:00'::timestamp)
Planning Time: 5.161 ms
```

Planning JIT Compilation – Example EXPLAIN

```
JIT:
  Functions: 9
  Generation Time: 5.174 ms
  Inlining: true
  Inlining Time: 75.291 ms
  Optimization: true
  Optimization Time: 124.676 ms
  Emission Time: 122.556 ms
```

Good Cases / Bad Cases

- CPU bound → likely good
  - JIT compilation can **only** help alleviate CPU usage
- OLTP / short query → bad
  - Overhead of JITing too high, bottlenecks elsewhere
- IO bound → not necessarily good, often not harmful
  - IO isn’t accelerated by JIT
- OLAP / long query→ good
  - Overhead of JITing lower portion of time, more likely to have CPU intensive aggregates
- wide relations → good
  - tuple deforming benefits, especially with lots of NOT NULL columns

Good Cases / Bad Cases

- Sequential Scans → helpful
  - tuple deforming and quals
- Index Scans → not helpful
  - nothing to accelerate, if no filter
  - indexing code often majority of time
- Bitmap Index Scans → helpful
  - tuple deforming and recheck
- Hash / Group Aggregate → helpful
  - nearly all of work JITed
  - aggregation work accelerated, but sorting is not
- Sort → not helpful
  - sorting not currently accelerated
- Joins → less helpful
  - join conditions often not accelerated (index nested loop, merge join, hashjoin)
  - non join quals & projection are accelerated

TPC-H Q01

```
Sort  (cost=4313208.98..4313209.00 rows=6 width=68)
      (actual time=26509.669..26509.670 rows=4 loops=1)
  Sort Key: l_returnflag, l_linestatus
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=1186602
  ->  HashAggregate  (cost=4313208.80..4313208.91 rows=6 width=68)
                     (actual time=26509.622..26509.625 rows=4 loops=1)
        Group Key: l_returnflag, l_linestatus
        Buffers: shared hit=1186602
        ->  Seq Scan on lineitem  (cost=0.00..1936377.20 rows=59420790 width=36)
                                  (time=0.016..8132.990 rows=59414087 loops=1)
              Filter: (l_shipdate <= '1998-09-18 00:00:00'::timestamp)
              Rows Removed by Filter: 571965
              Buffers: shared hit=1186602
Planning Time: 5.161 ms
Execution Time: 26509.857 ms
```

Identifying Queries / Workloads where helpful

- `top` → cpu bound or not
- `iostat -xm` → IO bound
- pg_stat_activity → queries run for long or not
- `track_io_timing = on`
- pg_stat_statements

  -
    ```
    SELECT blk_read_time, total_time, calls,
        total_time / calls AS avg_time,
        query
    FROM pg_stat_statements;
    ```
- EXPLAIN (ANALYZE, BUFFERS)
  - most of the time IO → probably not
  - most of time in “helpful” nodes → probably helpful
  - short → not helpful
  - really wrong costs → oops

Problems / Improvements: Query Planning

- Simplistic Costing
  - cost calculation constant – but actual time cost is not
  - `enable_*` GUCs wreak havoc, reach limit
  - might be cheaper overall to run plan “touching” more tuples after JITing
  - bad cost estimates → unnecessary JITing
- Whole Query decision too coarse
  - use estimates about total number of each function evaluation?
- JIT more aggressively when using prepared statements?
  - but ….

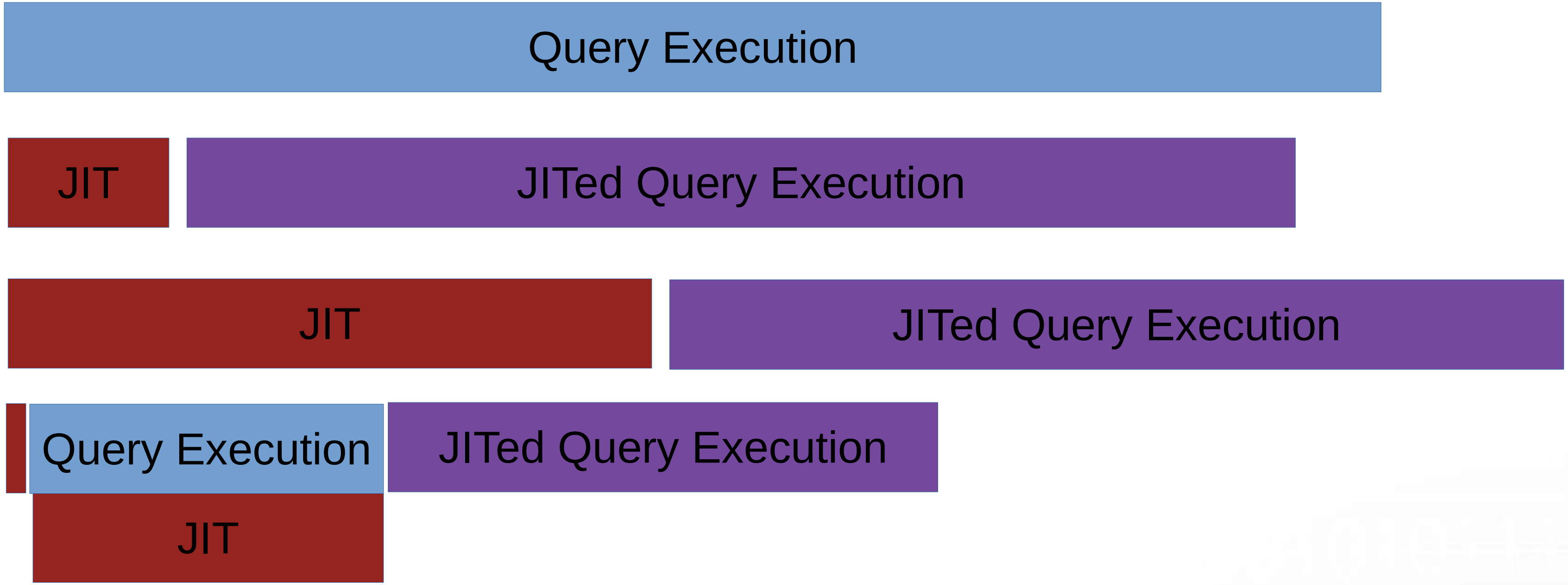
JIT Improvements: Caching

- Optimizer overhead significant
  - TPCH Q01: unopt, noinline: time to optimize: 0.002s, emit: 0.036s
  - TPCH Q01: time to inline: 0.080s, optimize: 0.163s, emit 0.082s
- Non-Shared / Shared / Persistent?
- But ...

JIT Improvements: Code Generation

- Expressions refer to per-query allocated memory
  - **Lots** of superflous memory reads/writes for arguments, optimizer can’t eliminate in most cases
    - massively reduces benefits of inlining
  - Optimizer can’t optimize away memory lots of memory references
  - FIX: separate permanent and per eval memory
- Expression step results refer to external memory by pointer
  - Move to on-stack allocation
- Function Call Interface references external memory
  - Move to on-stack allocation
  - Non JITed expression evaluation will benefit too
- Allows Caching, including sharing JITed functions between leader & worker
- Prototype: 2.2x improvement for TPC-H Q01

JIT Improvements: Incremental JITing



Future things to JIT

- COPY parsing, input / output function invocation
  - easy – medium
- Aggregate & Hashjoin hash computation
  - easy
- Tuple Sorting (in-memory)
  - including tuple deforming (from MinimalTuple)
  - easy
- Executor control flow
  - hard, but lots of other benefits (asynchronous execution, non-JITed will be faster, less memory)

JIT – how to test

- Debian:
  - apt.postgresql.org has v11 beta 3 packages
  - https://wiki.postgresql.org/wiki/Apt/FAQ#I_want_to_try_the_beta_version_of_the_next_PostgreSQL_release
  - Install postgresql-11
- RHEL:
  - yum.postgresql.org has v11 beta 3 packages
  - https://yum.postgresql.org/repopackages.php#pg11
  - install postgresql11-server postgresql11-llvmjit
    - depends on EPEL
- https://www.postgresql.org/docs/devel/static/jit.html
- Report Bugs & Problems!