State of JIT – 2018 Edition

Andres Freund

PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec
anarazel.de/talks/2018-06-01-pgcon-state-of-jit/state-of-jit.pdf

PostgreSQL 11 will have basic JIT compilation

v10+ Expression Evaluation Engine

- WHERE a.col < 10 AND a.another = 3
  - EEOP_SCAN_FETCHSOME (deform necessary cols)
  - EEOP_SCAN_VAR (a.col)
  - EEOP_CONST (10)
  - EEOP_FUNCEXPR_STRICT (int4lt)
  - EEOP_BOOL_AND_STEP_FIRST
  - EEOP_SCAN_VAR (a.another)
  - EEOP_CONST (3)
  - EEOP_FUNCEXPR_STRICT (int4eq)
  - EEOP_BOOL_AND_STEP_LAST (AND)
- direct threaded
- lots of indirect jumps

JITed expressions

- directly emit LLVM IR for common opcodes
- emit calls to functions implementing less common opcodes
  - can be inlined
- indirect opcode→opcode jumps become direct
- indirect funcexpr calls become direct
  - can be inlined
- TPCH Q01 non-jitted vs jitted:
  - 28759 ms vs 22309 ms
  - branch misses: 0.38% vs 0.07%
  - iTLB load misses: 58,903,279 vs 48,986 (yes, really)

Tuple Deforming

- deforming := turn on-disk tuple into in-memory representation
- Often most significant bottleneck
- TupleDesc (“tuple format”) can be made known at JIT time in many cases
- Optimizable:
  - Number of columns to deform - constant
  - Number of columns in tuple – if to-deform below last NOT NULL
  - column type - constant
  - column width – known for fixed width types
  - Variable alignment requirements – known for fixed width (depending on NULLness)
  - NULL bitmap – no need to check if NOT NULL
- Resulting code often very pipelineable, previously lots of stalls
- Access to tuple’s `t_hoff` / `HeapTupleHeaderGetNatts()` still major source of stalls
  - reorder tuple accesses on page!
- TPC-H Q01: unjitted deform vs jitted
  - time: 22277 ms vs 19580 ms
  - branches: 1396.318 M/sec vs 1161.628M/sec (despite higher throughput)

Good Cases / Bad Cases

- OLTP → bad, short query → bad
- OLAP → good, long query → good
- IO bound → not necessarily good
- CPU bound → likely good
- lots of aggregates → good
- wide relations → good

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;
```

```
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
```

EDB POSTGRES

Inlining

```
CREATE OPERATOR pg_catalog.= (
    PROCEDURE = int8eq,
    LEFTARG = bigint,
    RIGHTARG = bigint,
...
);
CREATE OR REPLACE FUNCTION pg_catalog.int8eq(bigint, bigint)
    RETURNS boolean
    LANGUAGE internal
    IMMUTABLE PARALLEL SAFE STRICT LEAKPROOF
AS $function$int8eq$function$
```

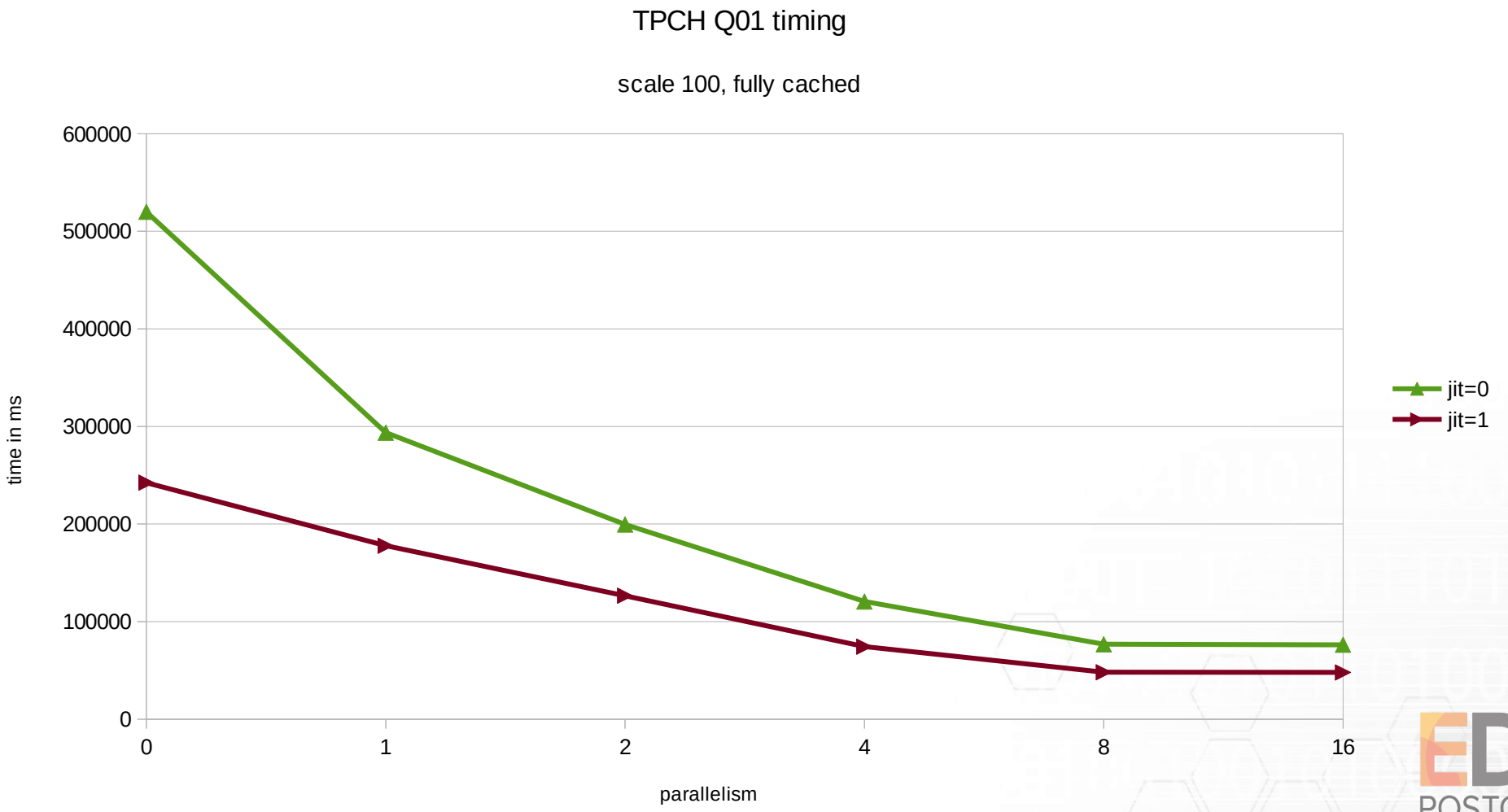
Inlining

- All operators in postgres are functions! Lots of external function calls
- Postgres function calls are expensive, lots of memory indirection
- Convert sourcecode to bitcode at buildtime, install into

  `$pkglibdir/bitcode/<module>.index.bc`

  `$pkglibdir/bitcode/<module>/path/to/file.bc`
- LLVM's cross-module inlining not suitable
  - requires exporting of symbols at compile time, unknown which needed
- Postgres specific inlining logic:
  - lookup symbol in summary corresponding to function
  - inlining safety check (no mutable static variables referenced)
  - cost analysis
  - inline function, referenced static functions, referenced constant static variables (mainly strings)
  - use `llvm::IRMover` to move relevant globals
  - can't cache modules in memory, cloning expensive and incomplete
- Avoids need to implement direct JIT emission for lots of semi critical code
- Function call interface significantly limits benefits

Faster Execution: JIT Compilation



Planning JIT

- Naive!
- Perform JIT if `query_cost > jit_above_cost`
- Optimize if `query_cost > jit_optimize_above_cost`
- Inline if `query_cost > jit_inline_above_cost`
- `-1` disables
- Whole query decision
- *NOT* a tracing JIT:
  - costing makes tracing somewhat superflous
  - tracing decreases overall gains

Switch to shell already

Profiling JIT

- Requires patches to LLVM, about to be integrated into LLVM trunk
  - debugger is same
- `jit_profiling_support = 1`
- Use:

```
perf record -k1 --call-graph lbr -p 7170 -o /tmp/perf.data
perf inject --jit -i /tmp/perf.data -o /tmp/perf.jit.data
perf report -i /tmp/perf.jit.data
```

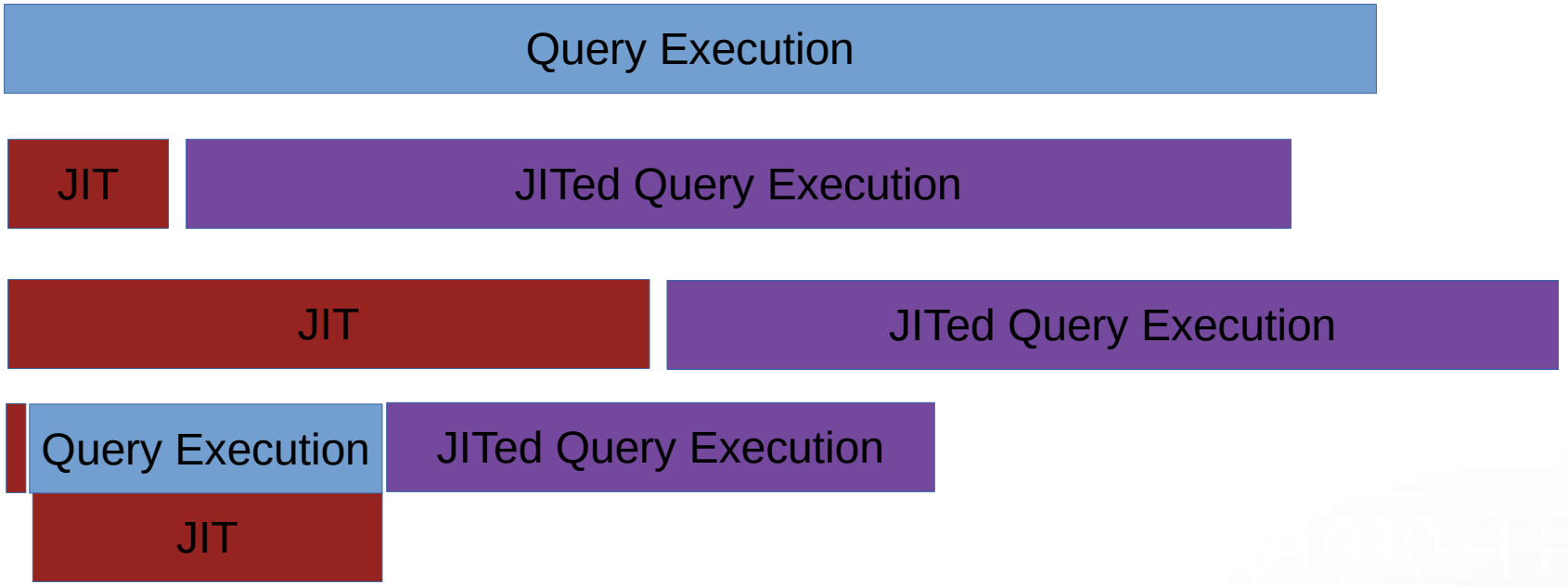
JIT Improvements: Code Generation

- Expressions refer to per-query allocated memory
  - generated code references memory locations
  - **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 persistent memory
  - move to temporary memory
- Function Call Interface references persistent memory
  - FIX: pass FunctionCallInfoData and FmgrInfo separately to functions
    - remove FunctionCallInfoData->flinfo
    - move context, resultinfo, fncollation to FmgrInfo
    - move isnull field to separate argument? Return struct?
  - Non JITed expression evaluation will benefit too

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
- references to memory locations prevent caching (i.e. expression codegen has to be optimized first)
- Introduce per-backend LRU cache of functions keyed by hash of emitted LRU (plus comparator)
  - What to use as cache key?
    - IR? - requires generating it
    - Expression Trees?
    - Prepared Statement?
- Shared / Non-Shared / Persistent?
- whole query decision – allows to eliminate redundancies, reduce mmap overhead, etc.
- relatively easy task, once pointers removed

JIT Improvements: Incremental JITing



JIT Improvements: Planning

- Whole Query decision too coarse
  - use estimates about total number of each function evaluation?
- Some expressions guaranteed to only be evaluated once
  - VALUES()
  - SQL functions
- JIT more aggressively when using prepared statements?

Future things to JIT

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

State of JIT – 2018 Edition

Andres Freund

PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec
anarazel.de/talks/2018-06-01-pgcon-state-of-jit/state-of-jit.pdf