Precision Performance Surgery for PostgreSQL

LLVM–based Expression Compilation, Just in Time

Dennis Butterstein Torsten Grust

Universität Tübingen,
Tübingen, Germany

[dennis.butterstein, torsten.grust]@uni-tuebingen.de

ABSTRACT

We demonstrate how the **compilation of SQL expressions into machine code** leads to significant query runtime improvements in PostgreSQL 9. Our primary goal is to connect recent research in query code generation with one of the most widely deployed database engines. The approach calls on LLVM to translate arithmetic and filter expressions into native x86 instructions just before SQL query execution begins. We deliberately follow a non-invasive design that does not turn PostgreSQL on its head: interpreted and compiled expression evaluation coexist and both are used to execute the same query. We will bring an enhanced version of PostgreSQL that exhibits notable runtime savings and provides visual insight into exactly where and how execution plans can benefit from SQL expression compilation.

1. WHAT TOOK YOU SO LONG, POSTGRESQL?

In a discussion of query processing strategies, the *evaluation of SQL expressions*—here we refer to expressions over scalar values, notably of number types as well as Booleans—typically assumes a second-tier role. Still, expression evaluation is pervasive in query plan execution: table scans, filters, aggregates, projections, and even joins (those which do not enjoy index support) inherently rely on it. Indeed, in the case of TPC-H [7], the inefficient evaluation of complex expressions has been identified as a major *choke point* [2, see choke point CP 4.1d "interpreter overhead"]. The premise of the present work is that significant query runtime improvements are obtained if we can speed up expression evaluation.

Expression Evaluation in the Limelight. Figure 1 shows query *Q1* of the TPC-H benchmark with a particular focus on the SQL expressions that are embedded in this query:

- a Boolean filter expression ① that compares values of type `date` (the `date` difference operator `-` is evaluated at query compile time and thus is of no concern in the context of this work) and

Proceedings of the VLDB Endowment, Vol. 9, No. 10


```
1  SELECT l_returnflag, l_linestatus,                                    ②
2         SUM(l_quantity) AS sum_qty,
3         SUM(l_extendedprice) AS sum_base_price,
4         SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
5         SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) AS sum_charge,
6         AVG(l_quantity) AS avg_qty,
7         AVG(l_extendedprice) AS avg_price,
8         AVG(l_discount) AS avg_disc,
9         COUNT(*) AS count_order
10   FROM lineitem
11   WHERE l_shipdate <= date '1998-12-01' - interval '103 days';  ①
12 GROUP BY l_returnflag, l_linestatus
13 ORDER BY l_returnflag, l_linestatus;
```

Figure 1: TPC-H *Q1*. We focus on the evaluation of SQL expressions (here: ① and ②) embedded in such queries.

- a group ② of aggregates whose arguments are arithmetic expressions over `double precision` columns and literals.

The execution of query *Q1* involves a substantial expression evaluation effort. Table 1 displays an excerpt of a function call profile, recorded while PostgreSQL 9 was executing *Q1* over a TPC-H instance of scale factor 5. We see that the scan of table `lineitem` leads to 29 999 799 invocations of function `ExecQual` to evaluate filter expression ① over the incoming rows (about 520 000 of those do not qualify such that `ExecScan` returns 29 477 776 times to its caller, each time delivering an individual qualifying row).[1] Each such row leads to the evaluation of the 8 aggregates and their arithmetic expressions ②, yielding a total of $8 \times 29\,477\,776 \approx 235\,582\,212$ invocations of `ExecProject`, the PostgreSQL function that evaluates expressions in a query's `SELECT` clause. Finally, caller `ExecAgg` returns 4 times[1] to deliver the rows of aggregates computed by *Q1*.

This expression evaluation workload has a measurable impact on query runtime. Indeed, in the case of *Q1*, PostgreSQL spends the lion share of the execution time on expression evaluation. The pie charts of Figure 2 detail this impact for an entire set of TPC-H queries (we have selected these queries because they embed several and/or complex SQL expressions—*Q19*, for example, contains a variety of filters, see Figure 5 below). Here, the darker pie slices account for the overall execution time spent in all functions in the call tree below `ExecQual` and `ExecProject`. During the execution of *Q1*, PostgreSQL is busy with expressions about $12.1\% + 39.8\% = 51.9\%$ of the time—for the further queries

[1] `ExecScan` and `ExecAgg` return one additional time only to indicate that no more rows will be delivered. PostgreSQL implements a Volcano-style iterator model [3].

Table 1: PostgreSQL execution profile, focus on the evaluation of the expressions ① and ② in *Q1* (see Figure 1). Functions under `ExecProcNode` comprise the expression interpreter (invoked 29 447 787 times).

# Calls	Function
29 447 787	ExecProcNode
5	├─ExecAgg
29 447 776	│ ├─advance_aggregates
235 582 212	│ │ └─**ExecProject**
58 895 550	② │ │ └─ExecMakeFunctionResultsNoSets
	② │ │ ├─ExecEvalConst
	② │ │ ├─ExecEvalScalarVarFast
	② │ │ ├─float8pl
	② │ │ ├─float8mul
	② │ │ ├─slot_getattr
	② │ │ └─ ···
235 582 208	│ ├─advance_transition_function
88 343 328	│ │ └─float8_accum
235 582 212	│ ├─slot_getsomeattrs
29 447 776	│ └─LookupHashTableEntry
176 686 640	│ └─ ··· slot_getattr
29 447 777	└─ExecScan
29 999 799	① └─**ExecQual**
29 999 794	① └─ExecMakeFunctionResultNoSets
	① ├─ExecEvalConst
	① ├─ExecEvalScalarVarFast
	① ├─date_le_timestamp
	① └─slot_getattr

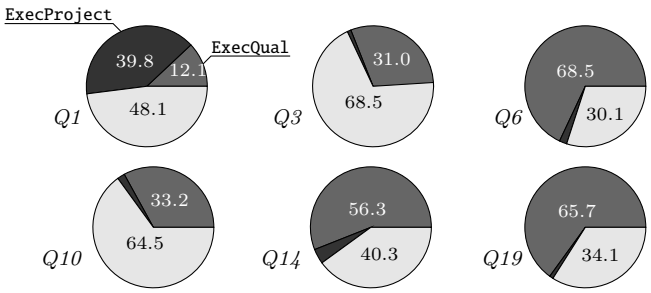

Figure 2: Percentages of overall execution time spent in **interpreted** arithmetic (`ExecProject`) and filter (`ExecQual`) expression evaluation for selected TPC-H queries.

in the set we observe that the system needs to devote between 32% and 70% of the query runtime to the evaluation of SQL expressions.

The Interpreter is Calling. Again. The PostgreSQL family of `Exec···` functions together form an *interpreter* that walks a tree-shaped representation of an expression: operator nodes hold a pointer to a function that, when invoked, will recursively evaluate subexpressions as well as the operator itself. The leaves of this tree represent literals (see `ExecEvalConst` in Table 1), row variables (`ExecScalarVarFast`), or column accesses (`slot_getattr`). While this style of expression interpreter is pervasive in today's database query processors, it has long been identified as CPU-intensive and outright wasteful on modern computing and memory architectures [1, 6]. Interpreter-induced function calls need to prepare/remove stack frames, save/restore registers, and jump to and from the diverse function bodies, leading to pipeline flushes and instruction cache pollution.

The resulting interpretation overhead is significant and may dominate all other tasks of the query processor. PostgreSQL's `EXPLAIN ANALYZE` output for *Q1* (Figure 3) reveals that the sequential scan of `lineitem` requires 6472 ms—3934 ms (12.1% of 32 516 ms, see Figure 2) of this time is spent in the interpreted evaluation of filter ①. In the 26 s between timestamps t_0 and t_1, PostgreSQL performs grouping and aggregation—the evaluation of the arithmetic expressions ② requires one half of this time (39.8% of 32 516 ms).

```
Sort (actual time=32516.050..32516.050 rows=4)
 Sort Key: l_returnflag, l_linestatus                    t1
 Sort Method: quicksort  Memory: 25kB
 -> HashAggregate (actual time=32495.849..32495.852)
   -> Seq Scan on lineitem (actual time=0.009..6472.002 rows=29447776)
      Filter: (l_shipdate <= '1998-08-20')                 ①
      Rows Removed by Filter: 552019                                t0
Total runtime: 32516.153 ms
```

Figure 3: Query plan and breakdown of the 32 516 ms elapsed execution time for *Q1* (`EXPLAIN ANALYZE` output).

2. COMPILATION OF SQL EXPRESSIONS

For any given expression e, at query run time the PostgreSQL interpreter will repeatedly walk the tree for e and invoke the same `Exec···` functions in the same order. The promise of **compiling SQL expressions** into machine code is to turn this repeated run time effort into a one-time compile time task. The present work is an exploration of how PostgreSQL can benefit if we trade expression interpretation for compilation. Cornerstones of the approach are:

- Each arithmetic and filter expression e is seen as a unit that is compiled into a separate function—to invoke the evaluation of e, PostgreSQL will thus call a single function.
- The PostgreSQL query optimizer remains unchanged—expressions are compiled after planning and just before query execution starts.
- This just-in-time compilation of expressions is based on the LLVM compiler infrastructure [5] which comes in shape of a library that we link with the original PostgreSQL code—LLVM offers high-quality code generation at low compilation times.
- We adopt a non-invasive approach that—outside of expression evaluation—retains PostgreSQL's Volcano-style pipelining query processor [3].
- Compiled and interpreted expression evaluation coexist; both can contribute to the execution of the same query.
- Compiled code calls on built-in PostgreSQL routines to access columns or convert values—this ensures compatibility with vanilla PostgreSQL and aids rapid prototyping. Such routines can be gradually reimplemented in terms of LLVM code if desired.

Our overall goal is to connect recent research in query code generation with the internals of a database system that sees world-wide deployment.

2.1 Compiling with Holes

To provide an impression of the compilation scheme, let us focus on the treatment of conjunctions and disjunctions in filters. This still grants insights into general efficiency considerations, in particular the economy of column access.

Figure 4(a) (left-hand column) shows the LLVM pseudo code that is emitted for the conjunctive filter expression $e \equiv p_1(\texttt{A})\ \texttt{AND}\ p_2(\texttt{B})$. Here, $p_1(\texttt{A})$ is an arbitrary filter expression that reads column `A`. In the code, `%`r denotes LLVM register r

$e \equiv p_1(\texttt{A})$ AND $p_2(\texttt{B})$ | e OR $p_3(\texttt{A},\texttt{B})$

```
    %a  = ⟨slot_getattr(A)⟩
    %p1 = ⟨p1(%a)⟩
⊛   br %p1, label %l0, label %l2
%l0: %b  = ⟨slot_getattr(B)⟩
    %p2 = ⟨p2(%b)⟩
    br %p2, label %l1, label %l2
%l1: Ⓣ
%l2: Ⓕ
```

plugs into Ⓣ:

```
ret true
```

---

plugs into Ⓕ:

```
%b  = ⟨slot_getattr(B)⟩
%p3 = ⟨p3(%a,%b)⟩
ret %p3
```

(a) Compiling filter subexpressions using continuation holes Ⓣ/Ⓕ: code plugged into hole Ⓣ may assume that e has evaluated to true (likewise for Ⓕ/false). Note that hole Ⓕ at label %l2 may be reached via two code paths.

$e \equiv p_1(\texttt{A})$ AND $p_2(\texttt{B})$ | e OR $p_3(\texttt{A},\texttt{B})$

```
    %a  = ⟨slot_getattr(A)⟩
    %p1 = ⟨p1(%a)⟩
    br %p1, label %l0, label %l2
%l0: %b  = ⟨slot_getattr(B)⟩
    %p2 = ⟨p2(%b)⟩
    br %p2, label %l1, label %l3
%l1: Ⓣ
%l2: Ⓕ1
%l3: Ⓕ2
```

Ⓣ with $R = \{\texttt{A} \mapsto \texttt{\%a}, \texttt{B} \mapsto \texttt{\%b}\}$:

```
ret true
```

Ⓕ1 with $R = \{\texttt{A} \mapsto \texttt{\%a}\}$:

```
%b  = ⟨slot_getattr(B)⟩
%p3 = ⟨p3(%a,%b)⟩
ret %p3
```

Ⓕ2 with $R = \{\texttt{A} \mapsto \texttt{\%a}, \texttt{B} \mapsto \texttt{\%b}\}$:

```
%p3 = ⟨p3(%a,%b)⟩
ret %p3
```

(b) Code emitted once hole Ⓕ has been split into $Ⓕ_{1,2}$.

Figure 4: Expression compilation: LLVM pseudo-code emitted for the evaluation of the filter $\big(p_1(\texttt{A})$ AND $p_2(\texttt{B})\big)$ OR $p_3(\texttt{A},\texttt{B})$.

(of which there are arbitrarily many—these will be mapped onto real CPU registers by code generation). $\langle p_1(\texttt{\%a})\rangle$ stands in for the LLVM code for p_1, assuming that the value of column A is available in register %a. Finally, $\langle\texttt{slot\_getattr}(\texttt{A})\rangle$ represents the LLVM instructions needed to invoke PostgreSQL's built-in routine that extracts the value of column A from the current row.

We see that the first branch instruction br (marked ⊛ in Figure 4(a)) implements Boolean shortcut: if the value of $p_1(\texttt{A})$, held in register %p1, turns out to be false, we ignore $p_2(\texttt{B})$ and immediately branch to label %l2. The *false hole* Ⓕ defines a spot where we can plug in continuing code [4]. Execution reaches the *true hole* Ⓣ at label %l1 only if both $p_1(\texttt{A})$ and $p_2(\texttt{B})$ evaluate to true.

Code that plugs into hole Ⓣ (Ⓕ) may be generated under the assumption that subexpression e evaluated to true (false). We exploit this when we generate code for a containing expression like e OR $p_3(\texttt{A},\texttt{B})$, see Figure 4(a) (right-hand column). According to the semantics of disjunction, there is thus nothing left to do in hole Ⓣ and we immediately return via ret. At Ⓕ, however, the overall result depends on $p_3(\texttt{A},\texttt{B})$. We know that column A is definitely available in register %a but we cannot tell for column B: *two code paths* lead to hole Ⓕ at label %l2 and only on one has %b been assigned the value of $\langle\texttt{slot\_getattr}(\texttt{B})\rangle$. We thus need to play safe and perform column extraction for B in any case. This is unfortunate since calls to slot_getattr are costly: the routine (1) checks whether the column has already been extracted and thus cached, (2) retrieves the external column representation either from the cache or the row at the correct offset, and then (3) transforms the value to an internal main-memory representation.

Hole Splitting. The cost of slot_getattr motivates an improved compilation scheme that *uses holes to encode exactly which column values are present in what registers* when execution reaches a hole. In the case of our filter expression e this leads to a split of the false hole into $Ⓕ_1$ and $Ⓕ_2$ (Figure 4(b), left-hand column). At $Ⓕ_1$ (label %l2) we know that e evaluates to false and that %a holds column A, at $Ⓕ_2$ we additionally know that column B is present in %b. We can make good use of this and judiciously omit the $\texttt{slot\_getattr}(\texttt{B})$ call in hole $Ⓕ_2$. To issue the minimum number of column loads that need to happen in a specific hole, the expression translation maintains a compile-time mapping R of columns to LLVM registers (see Figure 4(b), right-hand column).

```
1  SELECT SUM(l_extendedprice*(1-l_discount)) AS revenue ③
2    FROM lineitem, part
3   WHERE p_partkey = l_partkey ④
4     AND p_size >= 1 ⑤
5     AND l_shipmode IN ('AIR','AIR REG') ⑥
6     AND l_shipinstruct = 'DELIVER IN PERSON')
7     AND (   p_brand = 'Brand#31'
8             AND p_container IN ('SM CASE','SM BOX',...)
9             AND l_quantity >= 4 AND l_quantity <= 14
10            AND p_size <= 5
11        OR p_brand = 'Brand#52'
12            AND p_container IN ('MED CASE','MED BOX',...)
13            AND l_quantity >= 12 AND l_quantity <= 22
14            AND p_size <= 10
15        OR p_brand = 'Brand#31'
16            AND p_container IN ('LG CASE','LG BOX',...)
17            AND l_quantity >= 29 AND l_quantity <= 39
18            AND p_size <= 15); ⑦
```

Figure 5: Once *Q19* has been optimized, PostgreSQL's interpreter effectively evaluates the highlighted expressions.

Since hole splitting effectively unfolds all possible code paths through a filter expression at compile time, we pay for this optimization in terms of code size. For TPC-H query *Q19* featuring complex predicates (see Figure 5), we indeed find that we now generate about 9 times as many LLVM instructions (expression ⑦ yields 156 code paths). Since SQL expressions are super-brief if compared to general-purpose programs, we are nevertheless ready to accept this size increase in order to reap the potential runtime savings.

2.2 The Bottom Line: Performance Gains

We set out to shift effort from query run time to compile time. This pays off only if the added compilation time does not eat up the performance gains. With LLVM, we measure translation times of no more than 40 ms when we handle TPC-H queries. Hole splitting adds to this but only moderately so: for *Q19* we see an increase of about 30%—this is still negligible for OLAP-class queries. The more rows a query processes, the more worthwhile expression compilation becomes.

Figure 6 documents the performance gain of expression compilation when PostgreSQL 9 processes a TPC-H benchmark of scale factor 5 (average of 10 runs reported). We see a

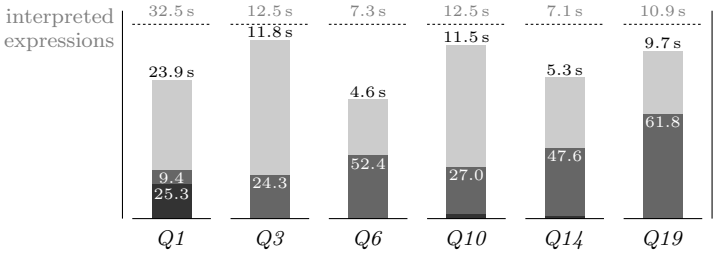

Figure 6: Percentages of overall execution time spent to evaluate **compiled** arithmetic and filter expressions (interpreted: see Figure 2). After compilation, TPC-H query *Q1* executes in 23.9 s (before: 32.5 s).

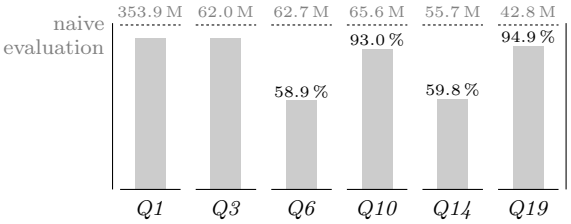

Figure 7: Reduction of the number of calls to `slot_getattr` (column value extraction) after hole splitting.

query runtime reduction of up to to 37% (*Q6*) for the family of selected TPC-H queries—in fact, all TPC-H queries exhibit performance improvements. The system now devotes a smaller slice of its time to expression evaluation: for *Q1*, SQL expressions now account for 9.4% + 25.3% = 34.7% of the overall effort (formerly: 51.9%, compare to Figure 2).

Figure 7 contains evidence that queries do benefit from hole splitting if an embedded expression repeatedly refers to the same set of columns. Even moderate repetition suffices to cut down the number of `slot_getattr` calls significantly: the filter expressions in *Q6* as well as *Q14* access columns `l_shipdate` and `l_discount` twice. No such column reuse within one expression occurs in *Q1* or *Q3*. Expression ⑦ of *Q19* (Figure 5) is a prime candidate for hole splitting—it is because of the high selectivity of the conjuncts ④ to ⑥ that we only measure a minor runtime impact: the native code for ⑦ needs to be hardly ever invoked by PostgreSQL.

3. DEMONSTRATION SETUP

We will bring an installation of PostgreSQL (version 9) that has been enhanced with an LLVM-based compiler for arithmetic and Boolean expressions, as described in Section 2.1. The on-site demonstration features a setup chosen to provide cursory as well as deeper impressions of SQL expression compilation:

Cursory. Our PostgreSQL 9 system comes with a visual `EXPLAIN` plan renderer (see Figure 8) that helps to understand how the system spends its time. Colored operator labels, like largest or slowest, let performance choke points stick out even if plans get complex. Paired execution time annotations (after|before) give a quick overview of what is to be gained by SQL expression compilation for a particular query. Additionally, we have instrumented PostgreSQL's

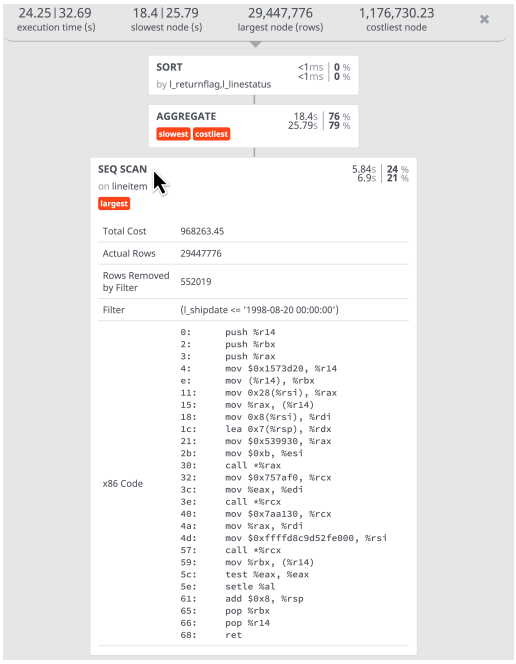

Figure 8: Enhanced visual `EXPLAIN`, revealing the LLVM-generated x86 instructions that implement the filter expression ① (cf. Figure 1). Plan rendering based on `Pev`.[2]

query processor such that execution time breakdowns in the form of pie charts (after: ◔, before: ◕, recall Figure 2) can be output on the fly.

Deeper. On a click, `EXPLAIN` reveals the LLVM intermediate representation [5] and/or the native x86 instructions for any expression that underwent compilation. Among other gory details, this also shows how hole splitting shapes the generated code. A larger TPC-H instance will be preloaded to demonstrate the runtime savings we have reported here. The demonstration does *not* run on rails, though: we will also provide toy data sets that allow for quick turnaround and experimentation. The audience is encouraged to explore ad-hoc query compilation scenarios and observe the impact of this PostgreSQL performance surgery.

4. REFERENCES

[1] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, pages 225–237, 2005.

[2] P. A. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proc. TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*, pages 61–76, 2013.

[3] G. Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE TKDE*, 6(1):120–135, 1994.

[4] D. Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, New Jersey, USA, 1971.

[5] *The LLVM Compiler Infrastructure Project*. `llvm.org`.

[6] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *Proc. VLDB*, pages 539–550, Seattle, Washington, USA, 2011.

[7] *The TPC Benchmark H*. `tpc.org`.

[2] `github.com/AlexTatiyants/pev`