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

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

<table>
<thead>
<tr>
<th># Calls</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 447 787</td>
<td>ExecProcNode</td>
</tr>
<tr>
<td>5</td>
<td>ExecAgg</td>
</tr>
<tr>
<td>29 447 776</td>
<td>advance_aggregates</td>
</tr>
<tr>
<td>235 582 212</td>
<td>-ExecProject</td>
</tr>
<tr>
<td>58 895 550</td>
<td>-ExecMakeFunctionResultsNoSets</td>
</tr>
<tr>
<td>②</td>
<td>-ExecEvalConst</td>
</tr>
<tr>
<td></td>
<td>-ExecEvalScalarVarFast</td>
</tr>
<tr>
<td></td>
<td>-float8pl</td>
</tr>
<tr>
<td></td>
<td>-float8mul</td>
</tr>
<tr>
<td></td>
<td>-slot_getattr</td>
</tr>
<tr>
<td>235 582 208</td>
<td>... advance_transition_function</td>
</tr>
<tr>
<td>88 343 328</td>
<td>-float8_accum</td>
</tr>
<tr>
<td>235 582 212</td>
<td>-slot_getsometr</td>
</tr>
<tr>
<td>29 447 776</td>
<td>-LookupHashTableEntry</td>
</tr>
<tr>
<td>①</td>
<td>ExecScan</td>
</tr>
<tr>
<td>29 999 799</td>
<td>ExecQual</td>
</tr>
<tr>
<td>29 999 794</td>
<td>-ExecMakeFunctionResultNoSets</td>
</tr>
<tr>
<td></td>
<td>-ExecEvalConst</td>
</tr>
<tr>
<td></td>
<td>-ExecEvalScalarVarFast</td>
</tr>
<tr>
<td></td>
<td>-date_le_timestamp</td>
</tr>
<tr>
<td></td>
<td>-slot_getattr</td>
</tr>
</tbody>
</table>

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).

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 $e ≡ p_1(A)$ AND $p_2(B)$. Here, $p_1(A)$ is an arbitrary filter expression that reads column $A$. In the code, $%r$ denotes LLVM register $r$.
Figure 4: Expression compilation: LLVM pseudo-code emitted for the evaluation of the filter \((p_1 \land p_2)\) OR \(p_3\).

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

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
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 calls to \texttt{getattr} (column value extraction) after hole splitting.

query processor such that execution time breakdowns in the form of pie charts (after: \textcircled{a}, before: \textcircled{b}, recall Figure 2) can be output on the fly.


deep.

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


\texttt{getattr} needs to be hardly ever invoked

\texttt{discount}

of that we only measure a minor runtime impact. Colored operator labels, like \texttt{SEASON} or \texttt{LARGEST}, help 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

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:

\textbf{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 \texttt{SEASON} or \texttt{LARGEST}, 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

\begin{itemize}
  \item \texttt{GET}\texttt{\_}\texttt{ATTR}\texttt{\_}\texttt{ATTR}
  \item \texttt{LARGEST}
  \item \texttt{SEASON}
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{Expression} & \textbf{Q1} & \textbf{Q3} & \textbf{Q6} & \textbf{Q10} & \textbf{Q14} \\
\hline
\textbf{Interpreted} & 23.9 s & 11.3 s & 4.6 s & 11.5 s & 5.3 s \\
\hline
\textbf{Naive} & 353.9 M & 62.0 M & 62.7 M & 65.6 M & 55.7 M \\
\hline
\end{tabular}
\caption{Comparison of \texttt{getattr} (column value extraction) after hole splitting.}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.png}
\caption{Percentage of overall execution time spent to evaluate \texttt{compiled} arithmetic and filter expressions (interpreted: see Figure 2). After compilation, TPC-H query Q1 executes in 23.9 s (before: 32.5 s).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{Reduction of the number of calls to \texttt{getattr} (column value extraction) after hole splitting.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8.png}
\caption{Enhanced visual EXPLAIN, revealing the LLVM-generated x86 instructions that implement the filter expression \texttt{Q1} (cf. Figure 1). Plan rendering based on \texttt{Pev}.\textsuperscript{2}}
\end{figure}

\texttt{getattribute}\texttt{\_}\texttt{ATTR}\texttt{\_}\texttt{ATTR}