Functional-Style SQL UDFs With a Capital ‘F’

Christian Duta Torsten Grust
University of Tübingen
Tübingen, Germany
[christian.duta, torsten.grust]@uni-tuebingen.de

ABSTRACT
We advocate to express complex in-database computation using a functional style in which SQL UDFs use plain self-invocation to recurse. The resulting UDFs are concise and readable, but their run time performance on contemporary RDBMSs is sobering. This paper describes how to compile such functional-style UDFs into SQL:1999 recursive common table expressions. We build on function call graphs to build the compiler’s core and to realize a series of optimizations (reference counting, memoization, exploitation of linear and tail recursion). The compiled UDFs evaluate efficiently, challenging the performance of manually tweaked (but often convoluted) SQL code. SQL UDFs can indeed be functional and fast.

ACM Reference Format:

1 RECURSION CLOSE TO THE DATA
Move your computation close to the data! This age-old mantra of the database community [34] asserts that we can expect a SQL query engine with immediate access to the data and its indexes to perform significantly better than an external processor to which we have to ship the data first. Indeed, if the computation exhibits a query-like style—and thus primarily filters, recombines, groups, and aggregates data—the lore holds up. But what if the computation is complex and deviates from common SQL query shapes, in particular (iterative or) recursive algorithms over tabular data as they are pervasive in, e.g., graph processing or machine learning [8, 13]?

As one example of such recursive computation, consider dynamic time warping (DTW) which is widely used in machine learning approaches to time series classification [14] and a variety of further domains. Given two time series \( X = (x_i)_{i=1..n} \) and \( Y = (y_j)_{j=1..m} \), DTW measures the distance between \( X \) and \( Y \) if we stretch (or compress) them along the time axis to align both optimally [4]. Its textbook-style recursive formulation reads:

\[
dtw(0,0) = 0 \\
dtw(i,0) = dtw(0,j) = \infty \\
dtw(i,j) = |x_i - y_j| + \min \{ dtw(i-1,j-1), dtw(i-1,j), dtw(i,j-1) \}. \tag{DTW}
\]

Figure 1 shows tabular encodings of \( X, Y \) and also illustrates how DTW maps (“warps”) the series’ elements onto each other.

Functional-style UDFs. With tables \( X \) and \( Y \) of Figure 1(b) in place, one possible in-database formulation of algorithm DTW is the recursive SQL UDF \( \text{dtw} \) of Figure 2. The body of UDF \( \text{dtw} \) resembles algorithm DTW as closely as SQL syntax would allow it. In particular,

- the recursion in DTW directly maps to recursive calls of \( \text{dtw} \) (the call sites are marked \( \text{rec} \) to \( \text{rec} \) in Figure 2), and
- the three-way case distinction in DTW manifests itself as a SQL CASE expression in \( \text{dtw} \).

![Figure 1: Time series X, Y, and their tabular encodings. The path of 0s in matrix \( \text{dtw}(i,j) \) indicates how to warp the series (e.g., \( x_i \) warps to \( y_j \), shown as ---- in (a)) for an overall distance of \( \text{dtw}(5, 5) = 0 \).]
Recursive self-invocation and function definition by case distinction are staples of a functional style of programming as it is used in, e.g., Haskell [20]. We thus refer to \texttt{dtw} in Figure 2 as a functional-style UDF.

Unfortunately, RDBMSs tend to penalize the functional UDF style at query run time or make its use practically impossible. PostgreSQL, for example, re-plans the body of a recursive UDF anew every time the concrete arguments of its next invocation are known [29, §38.5]. Even if body plans were cached, plans would need to be re-instantiated on every function call and later teared down on function return [29, §43]. Microsoft SQL Server [35] and Oracle [27] restrict UDF recursion depth to 32 and 50, respectively, which renders the functional style impractical from the start. MySQL generally disallows recursive self-invocation in SQL UDFs [23, §24.2.1].

UDFs and their invocation can be such a pain point for contemporary DBMSs that it is common developer wisdom to avoid them at all cost. Recent research has indeed aimed to get rid of functions at run time altogether [11, 18, 33].

Yet, our recursion-centric style incurs lots of function calls. The naïve evaluation of a call \texttt{dtw}(i,i) with \(i > 0\) leads to about \(0.87 \times (5.83^i / \sqrt{i})\) invocations of the function’s body [15]. A trace of PostgreSQL 11.3 in fact finds \texttt{dtw}(2,2) to plan and run the UDF’s body exactly 19 times; \texttt{dtw}(10,10) does so 12,146,179 times. Given this, the steeply growing evaluation time of \texttt{dtw}, shown as \(\bullet\) in Figure 3, does not surprise: PostgreSQL spends 72% of this time on query planning. (This and all following experiments were performed with PostgreSQL 11.3 running on a 64-bit Linux x86 host with 8 Intel Core™ i7 CPUs clocked at 3.66 GHz and 64 GB of RAM, 128 MB of which were dedicated to the database buffer. Timings were averaged over 10 runs, with worst and best run times disregarded.)

Compiling functional-style UDFs to recursive CTEs. \(\triangledown\) We tackle the disappointing performance of recursive SQL UDFs and develop a compilation technique that translates these functions into equivalent, efficient recursive common table expressions (CTEs) [28, 36]. The function compiler is realized as a SQL-to-SQL source translation that does not invade the underlying RDBMS. The present work thus enables SQL UDFs in functional style on all RDBMSs mentioned so far, even if those systems do not natively support such recursive functions or even lack any UDF facility. Once compiled, no traces of the original UDF, say \(f\), need to remain. The emitted CTEs may either (1) serve as a replacement for the recursive body of \(f\) or (2) be fully inlined into the SQL query that invokes \(f\) such that both can be optimized and planned in tandem.

For \texttt{dtw}, a basic implementation of this compilation strategy can already generate code (see \(\diamond\) in Figure 3) that surpasses the performance of a purpose-built CTE (see \(\square\)) and rivals that of a specifically optimized variant of the same \(\triangledown\). Hand-crafting and tweaking such CTEs tends to lead to complex code that is far from the original algorithm (DTW in this case).

{
  CREATE FUNCTION \texttt{dtw}(i int , j int ) RETURNS real AS $$
  \quad \text{WHEN } i=0 \text{ AND } j=0 \text{ THEN } 0.0
  \quad \text{WHEN } i=0 \text{ OR } j=0 \text{ THEN } \text{'Infinity'}:real
  \quad \text{ELSE (SELECT abs(X.x - Y.y)}$
  \quad \text{LEAST($\text{dtw}(i-1, j-1),}$
  \quad \text{dtw}(i-1, j ),$
  \quad \text{dtw}(i , j-1))$
  \quad \text{FROM X, Y }$
  \quad \text{WHERE } (X.t,Y.t) = (i,j))$
  \quad END;$$
  \text{LANGUAGE SQL STABLE STRICT;}
$$

Figure 2: DTW as a recursive SQL UDF written in functional style. \(\bigtriangleup\), \(\blacktriangleleft\), and \(\blacktriangledown\) mark the recursive call sites, \(\blacktriangleleft\) and \(\blacktriangledown\) designate the non-recursive base cases.

\(\triangledown\) Figure 3: Run time of implementations of \texttt{dtw}(i,i), measured on PostgreSQL 11.3 (logarithmic scales). This work aims for the grey area of low run times.

Call graphs as data. Using function compilation to free developers from the need to come up with hand-crafted CTEs is a first step that we describe in Sections 2 and 3. The ultimate goal is to improve the compiler such that the execution times of generated CTEs fall into the area \(\bigtriangleup\) of Figure 3. In Section 4, we develop such improvements all of which rely on an explicit, tabular representation of the UDF’s call graph. See Figure 4(a) for the call graph of \texttt{dtw}(2,2).

Sharing. An edge \(x \rightarrow y\) in the call graph for recursive function \(f\) indicates that the evaluation of \(f(x)\) has led to a recursive call \(f(y)\) at call site \(y\). Recursive calls that are shared by multiple invocations of \(f\) like \((1,1), (0,1),\) and \((1,0)\) in Figure 4(a)—indicate the potential to avoid repeated computation. Such sharing can drastically reduce the call graph size and thus evaluation effort, in the case of \texttt{dtw}(i,i) from \(O(5.83^i)\) down to \(O(i^2)\). The CTEs generated
The generated code operates in two phases in which the optimization opportunities which we explore in Section 4.3. This presents additional compile and memoization when we introduce both in the sequel. If we actual function evaluation. We quantify the impact of sharing as far as we can tell.

2 FUNCTION CALL GRAPH CONSTRUCTION AND EVALUATION

Let $f$ be a SQL UDF in functional style with scalar return type $r$ ($r$ is real for our example $dtw$). Recursion is expressed in terms of self-invocation of $f$ at, in general, several call sites (cf. to in the body of $dtw$ in Figure 2).

The compilation of $f$ replaces its body with SQL code that will evaluate a call, say $f(args)$, in two steps:

(1) **Construct call graph** $g$ that originates in root $args$ and records the arguments of all recursive calls that $f$ would perform. Since we do not evaluate these calls yet, $f$’s recursive calls may only depend on $args$ and any other database-wide accessible data, but not on $f$’s return values. The leaves of $g$ are the non-recursive base cases entered by $f$ (cf. $\tau$ and $\delta$ in the body of $dtw$).

(2) **Traverse** $g$ bottom up, evaluating the body of $f$ for the recorded arguments. Evaluating the body for root $args$ yields the overall result for the original call $f(args)$.

We elaborate on this two-step evaluation here and discuss its efficient SQL implementation in the subsequent Section 3.

The call graph provides us with an explicit run-time representation of the work that needs to be performed to evaluate $f(args)$. Figure 4(a) shows the graph we construct for a call $dtw(2,2)$.

*Edges in $\rightarrow$ site $\rightarrow$ out manifest that an invocation with arguments in leads $f$ to call itself at site $site$ with new arguments out. Refer to Figure 2 for $dtw$’s original body and its recursive call sites $\tau$ to $\delta$. Edges in $\rightarrow$ val towards a leaf val indicate that $f$’s caller enters a non-recursive base case that returns result value val of type $r$. In anticipation of our plan to construct call graphs using SQL, Figure 5 shows a straightforward tabular encoding call_graph of the graph in Figure 4(a). A call edge in $\rightarrow$ site $\rightarrow$ out is encoded as row $(in, site, fanout, out, val)$ in which fanout indicates that a call $f(in)$ leads to a total of fanout immediate recursive invocations; fanout $= 3$ characterizes $dtw$’s three-fold recursion. Likewise, base case edge in $\rightarrow$ val maps to row $(in, \Box, 0, in, val)$. We use $\Box$ to abbreviate SQL’s NULL.*

---

1. This is a syntactical restriction that may be sidestepped by writing $f$ in tail-recursive form. See Section 4.3 and Section 5.
call_graph(f, in, graph):
 1. calls ← [ ]
 2. FOR EACH call site of f that would recursively
   invoke f(out) if the arguments are in DO
   | calls[site] ← out
 3. edges ← \emptyset
 4. IF calls ≠ [] THEN
   | FOR EACH (site, out) IN calls DO
   |   ADD in ← site → out TO edges
   |   \textbf{val} ← evaluate body of f for arguments in
   |   ADD in ← val TO graph
   5. edges ← edges \setminus graph
   6. FOR EACH \textbf{call} IN edges DO
   |   ADD call_graph(f, out, graph ∪ edges) TO graph
 7. \textbf{return} graph

Figure 6: Call graph construction (pseudo code). Invoked via call_graph(f, in, ∅), returns a set of edges.

**Step 1: Call graph construction** can be described as a generic recursive process that accepts a function and arguments in. The pseudo code routine call_graph(f, in, graph) of Figure 6 does exactly that. Parameter graph, initially ∅, is used to accumulate the set of edges for the call graph of f(in).

We have already formulated this and following routines in a style that allows their direct transcription into SQL. You will find corresponding pseudo and SQL code regions in Section 3 to carry identical labels:

\textbf{slice calls} Given incoming argument in, we collect the arguments out of all immediate outgoing calls of f (if any) in associative array calls. To implement this, we embed a sliced version of f’s body into routine call_graph that computes out but does not perform recursive calls. We focus on slicing in Section 3.4.

\textbf{construct} If we have found that f(in) leads to outgoing calls, construct corresponding edges in ← out → using array calls. Otherwise, argument in led f into a base case: compute f’s return value val and add in ← val to the call graph.

\textbf{invoke} Continue call graph construction for any outgoing argument out we have not encountered earlier, accumulating constructed edges in set graph.

**Call sharing.** Note that call_graph(f, args, ∅) will construct a directed acyclic graph (or DAG) if the original UDF invocation f(args) terminates: a circular call graph would indicate a lack of recursion progress in f (which would thus loop indefinitely).

Most importantly, however, any node in (but the root) in the call graph may have an in-degree greater than one (see node (1,1) in Figure 4(a), for example). Since we assume f to be a pure function void of side effects, any call f(in) will yield the same computation. Node in, the sub-graph below it, and all evaluation effort for the sub-graph may thus be shared by all callers.

Routine call_graph implements this sharing through the accumulation of a set of edges. The space savings can be substantial, as Figure 7 shows. Call sharing leads the compiled SQL code to construct a call graph of (i + 1)² nodes for a call \texttt{dtw(i,i)}, see \textbf{\textbullet} in Figure 7. In contrast, recall our discussion in Section 1 in which we found PostgreSQL to not share the evaluation effort of individual calls (this applies even if f is explicitly marked as being free of side effects [29, §38.7]). Without sharing, the nine inner nodes of the \texttt{dtw(2,2)} call graph in Figure 4(a) would already unfold into a graph of 19 invocations. In general, PostgreSQL’s built-in function evaluation faces \texttt{dtw} call graphs of exponential size (\textbullet\textbullet\textbullet) which ultimately leads to disastrous function run times.

**Step 2: Call graph traversal (evaluation).** Under our new regime, the generated SQL code finalizes function evaluation via a traversal of f’s call graph. Figure 8(a) depicts this traversal for the sample call graph of \texttt{dtw(2,2)} shown in Figure 4(a).

The graph is traversed layer-by-layer, starting with the bottommost layer in which the call graph’s base case edges in ← val indicate that f(in) = val. We record these discoveries as rows (in, val) in the two-column table evaluation (see Figure 8(b)). This table is initially empty but will hold the results of all recursive function calls once evaluation is complete.

A call graph node in with n recursive calls, in ← out₁ → outₙ, becomes available for evaluation in the next higher layer once all n return values of these function calls are found in table evaluation, i.e., if \{(out₁, val₁), ..., (outₙ, valₙ)\} ⊆ evaluation. We then evaluate f for argument in using a simplified function body in which recursive call site \texttt{s} has been replaced by \texttt{valᵢ} (i = 1, ..., n). Evaluation of the body will return value \texttt{val}, which we enter as row (in, val) into table evaluation.

Figure 8 shows that this iterative evaluation process partitions the call graph for \texttt{dtw(2,2)} into four layers, traversed upwards from the leaves (dark to light). After the fourth iteration table evaluation holds row ((2,2), 1.0) which completes the evaluation with the final result \texttt{dtw(2,2)} = 1.0.

Routine evaluation(f, args, e, graph) of Figure 9 realizes this traversal for call graph graph with root node args. While we traverse graph, we use parameter e to accumulate the table of result rows. Initially, we expect e to only hold the
Let us close with a few notes on the merits of this call graph-centric approach to UDF compilation. Besides the opportunity to share calls and thus evaluation effort, we find:

- call graphs to be sufficiently general to represent $n$-way recursion (like the three-way recursion in DTW). Some functions lead to simpler, linear call graphs and we discuss how to exploit this in Section 4.3.
- Layer-based node scheduling uncovers independent calls: the for each in region body of routine evaluation can process all body evaluations of one layer in parallel.
- Further opportunities for parallel evaluation on a coarser level present themselves as call graphs with independent sub-graphs. (This is not pursued in the present paper.)
- Traversal-based evaluation creates a table filled with the results of all intermediate recursive calls. Future calls to $f$ can benefit if this table is kept around (see Section 4.2).

3 COMPILING FUNCTIONAL-STYLE UDFs TO RECURSIVE CTEs

We now describe the SQL-to-SQL compiler that translates a given functional-style UDF $f$ into recursive common table expressions. The compiled code is assembled from:

- two SQL code templates (transcriptions of the two routines call_graph and evaluation of Figures 6 and 9 from pseudo code to SQL),
- excerpts—so-called slices—of the original body of $f$ which we insert into those two templates.

The emitted code is entirely based on recursive CTEs and does not contain self-invocations of $f$.

We pursue a SQL source-to-source translation and thus expect the input UDF $f$ to adhere to the SQL dialect described by the grammar of Figure 11. Start symbol udf restricts our treatment to functions that:

- are free of side effects (in PostgreSQL, such UDFs may be tagged as STABLE or IMMUTABLE [29, §38.7], also see Section 4.2), and
- return values of some scalar type $\tau$.
The UDF’s body is formed by a top-level SELECT-FROM-WHERE block in which scalar and tabular subexpressions (cf. nonterminals e and f) may nest to arbitrary depth. The grammar distinguishes scalar expressions that may and may not contain self-invocations of f (non-terminals e and sql respectively). This already rules out some queries in which calls to f depend on each other. Ultimately, slicing (Section 3.4) will identify all queries that exhibit such problematic interdependencies.

Unique labels @ and ϕ are used to identify subexpressions, e.g., a function’s call sites. Labels are internal to the parse tree only.

### 3.1 SQL Template: Call Graph Construction

The recursive common table expression of Figure 12 computes the tabular encoding (recall Figure 5) of the call graph for f(ARGS). In SQL code templates, cursive type indicates template text that needs to be replaced. We use overlines to abbreviate comma-separated lists of columns. Further, for f(ARGS) ≜ dtw(i, j), f. ARGS denotes dtw.i, dtw.j.

To illustrate their workings, regions in the SQL code directly relate to those in the pseudo code of Figure 6:

**Invoke** Compute two-column table slices in which a row (i, out_i) indicates that the evaluation of f(ARGS) reaches call site s_i and would invoke f(out_i). If call site s_i is not reached for arguments ARGS, record (i, 1) in slices instead. Table slices will carry n rows if f has n recursive call sites (for dtw, n = 3 with s_i = @).

\[\text{Figure 11: A grammar for functional-style SQL UDFs. Expression labels (@, ϕ) are internal to the compiler.}\]

\[\text{Figure 12: Call graph construction (SQL template, compare with Figure 6). Note: code block A occurs twice.}\]

To obtain \(\text{out}_i\), we evaluate \(\text{slice}(f, s_i, [f.\text{ARGS}])\), the sliced variant of the body of f in which all subexpressions have been removed that are irrelevant to the evaluation of f’s argument \(\text{out}_i\); at call site \(s_i\). Slicing [38, 41] is an established code transformation technique that we adapt for SQL in Section 3.4. The result of \(\text{slice}([\text{dtw}, @, [i, j]])\) is shown in Figure 13.

**Table** For each recursive call site \(s_i\) that has been reached, collect (the tail of) its call graph edge \(\text{out}_i\) in table calls. We use window aggregate \(\text{COUNT}(\star)\) OVER () over the non-empty slices to find the number of recursive calls performed by f(ARGS) (recall our discussion of column fanout in Section 2).

**Construct** If, instead, arguments ARGS led to a base case (i.e., table calls is empty), evaluate the body of f for ARGS to obtain return value val. Construct (the tail of) the base case edge \(\text{val}\). We use \(\text{body}(f, [e_1, e_2, \ldots, [x_1, x_2, \ldots])\) to reproduce the body of UDF f in which call sites and arguments have been replaced by expressions \(e_i\) and \(x_j\), respectively. See Figure 14. Since the recursive call sites are irrelevant in a base case, the template sets \(e_i\) to NULL of type r.

Call and base case edges jointly form table edges that will be added to the call graph.

**Invoke** Add edges to the existing call graph. Proceed with call graph construction, now using the arguments \(\text{out}_i\) of the just added graph edges g as arguments to f. As per
**3.2 SQL Template: Call Graph Traversal**

The SQL template of Figure 15 realizes the layer-by-layer call graph traversal as introduced in Figure 9. Like the pseudo code, it returns binary table evaluation whose rows `(in, val)` indicate that `f(in) = val`.

This SQL piece assumes that (1) `f`'s return values for base cases are found in table `base_cases`, and (2) the tabular encoding of the call graph is in found table `call_graph`. 

**Schedule** Identify unevaluated nodes `g` whose recursive calls (of which there are `g.fanout` many) are all found in table evaluation. Collect the calls’ return values in SQL array `ret`.

**Body** For each such node `g`, evaluate the body of `f` with its call sites replaced by the return values found in `g.ret`. Record the found results in table `returns`(in, val). (As mentioned in Section 2, all of these body evaluations are independent and may be evaluated in parallel by the RDBMS.)

**traverse** Add returns to evaluation to form the overall known results so far. The CTE will continue to iterate until the result for argument `f.args` is indeed found in `results`.

---

```sql
WITH RECURSIVE evaluation(in,val) AS (  
    TABLE base_cases  
    UNION ALL  
    WITH RECURSIVE UNALL (  
        WITH e(in,val) AS (TABLE evaluation),  
        returns(in,val) AS (  
            SELECT go.in,  
                (body(f, [go.ret[1],...go.ret[n]]),  
                [(go.in), args])) AS val  
            FROM  
                select g.in, array_gather(e.val,g.site) AS ret  
                FROM  
                    call_graph AS g, e  
                WHERE  
                    g.out = e.in  
                AND  
                    NOT EXISTS (SELECT 1 FROM e WHERE e.in = g.in)  
                GROUP BY g.in, g.fanout  
                HAVING COUNT(*) = g.fanout  
            AS go(in,ret)  
            )  
        SELECT results.*  
        FROM  
            (TABLE e UNION ALL TABLE returns) AS results(in,val)  
        WHERE  
            NOT EXISTS (SELECT 1 FROM e WHERE e.in = ROW(f.args))  
        )  
    )  
    CREATE FUNCTION f(args) RETURNS r  
    AS $$  
    WITH RECURSIVE call_graph(in,site,out,val) AS (  
        (see Figure 12)  
    ),  
    base_cases(in,val) AS (  
        SELECT g.in, g.val  
        FROM  
            call_graph AS g  
        WHERE  
            g.fanout = 0  
    ),  
    evaluation(in,val) AS (  
        (see Figure 15)  
    )  
    SELECT e.val  
    FROM  
        evaluation AS e  
        WHERE  
            e.in = ROW(f.args);  
    $$  
)  
```

---

Figure 13: Slice of the body of UDF `dtw` for call site `s`. Subexpressions irrelevant to the computation of the arguments `i-1, j-1` at call site `s` have been removed.

```sql
body(dtw, [e1, e2, e3], [i, j]) =  
    SELECT CASE  
        WHEN i=0 AND j=0 THEN 1  
        WHEN i=0 OR j=0 THEN 1  
        ELSE (SELECT abs(X.x - Y.y) +  
            LEAST(e1,  
                e2,  
                e3))  
        FROM X, Y  
        WHERE (X.t,Y.t) = (i,j)  
    END;
```

---

Figure 14: Body of UDF `dtw` with its call sites and arguments replaced by `e1, e2, e3` and `i, j`, respectively.

---

Figure 15: Bottom-up call graph traversal (SQL template, compare with Figure 9).

---

Figure 16: Final compiled SQL code to replace the functional-style UDF `f`. (Compare with Figure 10.)
Figure 17: $q \rightarrow \pi$ derives the set $\pi$ of evaluation paths for SQL query $q$. Operator $\oplus$ combines two path sets: $\pi_1 \oplus \pi_2 = \{p_1 \parallel p_2 \mid p_1 \in \pi_1, p_2 \in \pi_2\}$ (denotes path concatenation).

Figure 18: (Prefix tree of) Evaluation paths of UDF $dtw$ superimposed on its body. $[\pi_1, \pi_2, \ldots]$ shown as $\pi_1 \rightarrow \pi_2 \rightarrow \ldots$.

exacts $f$’s return value for argument $\overline{\pi f}$ to deliver the function’s final result.

This purely CTE-based form of $f$ serves as a drop-in replacement for the original functional-style UDF.

3.4 Slicing for SQL

With $slice(f, s_i, \cdot)$ of Figure 12 we were after a “cut-down” version of $f$’s original UDF body in which only those expressions are retained that are relevant to the evaluation of the argument expression $out_i$ at call site $s_i$. It is crucial that $out_i$ evaluates the same in both, the original as well as the sliced body.

This closely resembles program slicing [41] in which a program is reduced to contain only those statements that are relevant to the execution of a given statement $s$ (the slicing criterion). Slicing has originally been introduced to aid program analysis and debugging [38, 40]. Here, we adapt slicing to functional-style SQL UDFs to aid their compilation to CTEs.

Evaluation paths. In the statement-by-statement execution of an imperative program, we can identify the trace of statements [3] that have been executed before slicing criterion $s$. In an expression-based language like SQL, an expression $e_1$ is entered before $e_2$ if an expression evaluator begins the evaluation of $e_1$ before it starts to evaluate $e_2$. This is the case if

- $e_2$ is a subexpression of $e_1$ (in this case, the evaluation of $e_2$ finishes before $e_1$), or
- $e_1$ binds a variable that is in scope in $e_2$, or
- $e_1$ is a predicate that may inhibit the evaluation of $e_2$.

Given a query $q$, we use $q \rightarrow \{p_1, \ldots, p_n\}$ to compute its set of evaluation paths. Each path $p_i$ is a sequence of expression labels $\{\pi_1, \ldots\}$ (see Figure 11) which stand in for their associated expressions $e_i$. Label $\oplus$ precedes $\oplus$ in $p_i$ (or: $\oplus < p_i$, $\oplus$), if $e_i$ is entered before $e_2$ in $q$. Figure 17 defines $\rightarrow$ in terms of inference rules that inspect the syntax of $q$.

Figure 18 superimposes the evaluation paths of UDF $dtw$ on its body query. We see that evaluation path $[\pi_1, \pi_2, \ldots]$ contains the expressions entered before base case literal 0.0 (label $\triangleright$) is evaluated. One evaluation path leading to call site $\triangleright$ is $p = [\pi_1, \pi_2, \ldots]$.

WHERE $e$

Figure 19 defines $\rightarrow$ in terms of inference rules that inspect the syntax of $q$.

Call site slices. We build on evaluation paths and define $slice(f, s_i, [x_1, x_2, \ldots])$ as follows:

1. Let $q$ denote the body query of UDF $f$. Derive its set $\pi$ of evaluation paths via $q \rightarrow \pi$. Let $\pi[s_i] \subseteq \pi$ hold the subset of paths that contain call site label $s_i$.

2. The labels in $C = \{s_i\} \cup \{c \mid c < p, s_i\}$ represent the expressions in $q$ that are entered before $s_i$ on some evaluation path. To guarantee that the resulting slice will not depend on other self-invocations of $f$, we impose the syntactic restriction that $C$ may not contain call site labels other than $s_i$.

3. Find the sliced query $q^\sim = q\|_C$ to remove expressions not relevant to the evaluation of the arguments at call site $s_i$. $\|_C$ is defined in Figure 19 and discussed below.
SQL transformation \( q\|_C \) is, again, defined by syntactic case analysis on \( q \). Set \( C \) guides the slicing which replaces irrelevant subexpressions by \( \top \), an arbitrary yet distinguished SQL value that SQL template \texttt{call_graph} uses to detect that the evaluation of \( q \) has not entered call site \( s_i \) (Figure 12, see \texttt{call_graph}). Notes on selected cases of \( q\|_C \) in Figure 19:

**Rec** If this is the call site we are after (\( \in C \)), remove the recursive call to \( f \) and package its \( n \) arguments using a row constructor. Otherwise, the expression is irrelevant.

**Op** If the call site is found in the \( n \)-th argument of an \( n \)-ary operator \( \otimes (e_i\|_C \neq \top \) and thus \( E = \{i\} \), keep slicing that argument \( e_i \). Otherwise discard the operator entirely.

**Sql** Do not descend further into scalar SQL expression \( sql \) as it will contain no call site (recall Figure 11).

**Case** Preserve the **Case** to ensure that branches \( e_1 \) and \( e_2 \) are (not) evaluated under the same conditions as in the original query. Recursive application of \( \|_C \) replaces irrelevant branches by \( \top \).

**Select** For a call site found in **Select** expression \( e_i \), keep the **From** and **Where** clauses as these bind in-scope row variables \( id_{21}, \ldots, id_{2k} \) and guard expression evaluation through predicate \( e \), respectively.

**Where** To slice for a call site inside a **Where** predicate \( e \), move \( e \) into the **Select** clause where we can slice \( e \) to extract and observe the call’s argument values.

Since the labels in \( C \) reflect the evaluation order of **Select-from-where** blocks (see Figure 17 for the \( \Rightarrow \) rule for such blocks), \( \in C \Rightarrow \in C \) and \( \not\in C \Rightarrow \not\in C \) (i.e., \( \Rightarrow \) is evaluated after **From** and **Select** is evaluated after **Where**). Rules **Select**, **Where**, **From** thus cover the possible slicings of the block.

**Figure 19:** \( q\|_C \) slices SQL query \( q \) to expose the arguments of the recursive call at the call site identified by \( C \).

(4) [Post-processing only.] Return \( q \) with \( f \)'s arguments replaced by expressions \( x_1, x_2, \ldots \).

4 COMPILER OPTIMIZATIONS

As Figure 3 indicated for \( dtw \), the performance difference between functional-style UDFs (\( -\bullet - \)) and their compiled counterpart (\( \bullet - \)) can already be drastic. Tweak to the vanilla compilation technique developed so far, let us enter the area of run times that can surpass manually crafted recursive CTEs. Below, we discuss three such tweaks and quantify their run time impact.

4.1 Evaluation with Reference Counting

In each iteration of a recursive common table expression WITH RECURSIVE \( t AS (q_1, UNION ALL q_2) \), query \( q_2 \) finds in \( t \) all rows that were produced in the CTE’s last iteration [36]. SQL implementations hold these newly found rows of \( t \) in the so-called work table [29, §7.8], ready to be read by \( q_2 \).

CTE evaluation of Figure 15 thus unions known and new results (see Line 18 in \texttt{traverse}) to ensure that \texttt{Schedule}
sees all results found so far: the evaluation of a call graph node may depend on a return value found in any lower layer of the graph traversal (cf. node (1, 2) in Figure 8(a) which depends on nodes (0, 2) and (0, 1) two layers down, for example). As evaluation goes on, this leads to monotonically increasing work table sizes which negatively affects CTE run time performance.

We observe that the in-degree of a call graph node determines how often its return value is referenced during the evaluation of parent calls. This suggests the following adaptation of the compilation scheme:

1. To the tabular encoding of the call graph, add extra column ref to hold the in-degree of each node. For dtw(2, 2), this augmented call_graph table is shown on the left (also see Figure 5).

2. Modify CTE evaluation: if a known return value has been referenced to construct a new result, place it in the work table with a decreased ref value. Should the ref value reach 0, drop the return value from the work table entirely.

Figure 20 traces the work table size as CTE evaluation traverses the call graph for dtw(100, 100). The vanilla CTE of Figure 15 indeed processes work tables of monotonically increasing size, growing from 201 to 10,201 rows across the 200 iterations. This incurs a noticeable runtime penalty for evaluation processes that recurse deeply (in Figures 20(a) and 20(b)). With reference counting, the work table size never exceeds 300 rows and decreases sharply as the traversal approaches the ever-narrower layers at the top of dtw’s call graph. These savings add up favorably at run time (see in both figures).

![Graph showing work table size and CTE run time](image)

(a) Work table shrinking. (b) Run time reduction.

Figure 20: Evaluating dtw(i,i): Impact of reference counting on work table size and CTE run time.

While sharing helps to keep work table sizes in check during call graph construction, reference counting does the same during evaluation. The evaluation of dtw(300, 300) (see Figure 23) builds a work table that never exceeds 901 rows if reference counting is performed. Even with a modest database buffer size of 128 MB (i.e., 0.2% of the host’s RAM of 64 GB), no buffer read or write I/O operations are performed by PostgreSQL.

4.2 Memoizing Return Values Across Calls

The result of call graph traversal is an entire table of function return values (recall Figure 8(b)). From this evaluation table, the compiled UDF f extracts the result associated with the call graph’s root args (see result in Figure 16). However, the return values of all intermediate recursive calls are just as precious, provided that

- we can expect f to be called many times (in a database setting where UDF invocations are embedded in queries, this would be the rule rather than the exception), and
- we know that f is referentially transparent [19], either generally or at least within a defined context (e.g., inside a transaction). In PostgreSQL, these degrees of referential transparency are declared via the function modifiers IMMTABLE or STABLE, respectively [29, §38.7].

Entries in table evaluation could then be used to accelerate the evaluation of future calls to f.

To implement this style of memoization [22] for f, we follow two simple steps:

1. After an evaluation of f, add the contents of evaluation to a table memo[in, val], discarding duplicate rows.

2. Upon subsequent invocations f(args), treat the entries in memo like additional base cases.

To implement this, use call_graph(f, args, memo) to construct the call graph (we used call_graph(f, args, ∅) before, cf. Figure 6).

In the constructed call graph, each such extra base case edge in val replaces an entire subgraph (with root in) whose recursive calls need not be evaluated since val is already available. Figure 4(b) shows the call graph for dtw(3, 3) which has been constructed based on the return values of an earlier dtw(2, 2) invocation. In this case, only the 7 calls at the fringes of the graph for dtw(3, 3) remain to be evaluated (down from 16 calls without memoization).

Note that this particular flavor of memoization already is beneficial if the memo table holds the root(s) of any subgraph of the current call graph [6]. In a sequence of invocations of f,
we can thus expect to start saving evaluation effort already early on. This is in contrast to plain memoization which only remembers the single return value at the root of an evaluated call graph [26]. Figure 22 plots the call graph sizes we observed during a sequence of 100 invocations of \( \text{dtw}(i, i) \) with random \( i \in \{1, \ldots, 100\} \). As expected, memoizing the top-most root call reduces calls graph sizes over time (\( \bigcirc \)). The memoization of subgraph roots (\( \bigcirc \)), however, is far more effective, bringing call graphs down to size 1 already after a dozen calls. (We repeated the random sequence of invocations multiple times and report average call graph sizes here.)

**Memoizing non-root calls.** Do functional-style SQL UDFs pave a way to efficient recursive computation close to the data? Our answer is the compiler of Section 3 with its reference counting and memoization optimizations enabled. Figure 23 reports a significant run time advantage for the compiled \( \text{dtw}(\bigcirc) \) over a hand-crafted CTE (\( \varnothing \)). The latter is slightly ahead for small time series lengths \( n \) where its comparatively short planning time of 0.5 ms pays off. The planning of the more complex SQL templates of the compiled \( \text{dtw} \) clocks in at 2.5 ms on PostgreSQL 11.3. Within reasonable effort, however, memoization can be added only for the root calls of this hand-crafted CTE: the return values of recursive calls are not generally available in the work table populated by a purpose-built CTE. Once we enter the area of calls with sizable recursion depth, \( i.e. \), time series of length \( n > 70 \) for which the complexity of DTW’s three-fold recursion becomes relevant (recall Figure 7), the compiled \( \text{dtw} \) with memo table thus has a much higher chance to benefit from earlier evaluation work—even if only partially. The compiled function scales to larger \( n \) (\( \bigcirc \)) in the same experiment.

**In-database compiled UDF vs. external processor.** For DTW, a database-external Python 3.7 function barely keeps up with the compiled recursive UDF. The Python-based implementation pays the price for data serialization and conversion to obtain access to the time series tables \( X \) and \( Y \) (recall Figure 1(b)) before it can perform the actual DTW computation (see \( \_\_\_\_\_\_\_¥ \) in Figure 23). Interestingly, we are absolutely required to write (or annotate) the recursive Python code to use memoization: plain Python constitutes a hopeless case (\( \varnothing \)).

**Wrapping memoization around recursive UDFs.** Memoization does not hinge on compilation. We can obtain \( f^{\text{memo}} \), a memoizing variant of the recursive SQL UDF \( f \), by wrapping \( f \)’s body as shown in Figure 24, a first-order SQL implementation of the memoization scheme described by Norvig [26]. Wrapper \texttt{Memoize} extracts \( f \)’s return value from table \texttt{memo} if present. Only if that fails, the original body of \( f \) is invoked (\texttt{blocks}). It is crucial that body recurses to \( f^{\text{memo}} \) to ensure that non-root recursive calls, too, are served from memo. \texttt{INSERT INTO}...\texttt{RETURNING val} maintains table \texttt{memo} as a side-effect of the evaluation of \( f^{\text{memo}} \) but ultimately returns \texttt{val} as expected by the caller. The function’s \texttt{VOLATILE} modifier ensures that inserts into \texttt{memo} are immediately visible once a recursive call returns [29, §37.7].

```
CREATE FUNCTION \( f^{\text{memo}}(\varnothing) \) RETURNS \( \tau \)
AS $$
\text{INSERT INTO} \text{memo}(\text{in}, \text{val})
\text{SELECT} m.* \text{FROM} \text{memo AS} m \text{WHERE} \text{ROW(f.\varnothing)} = m.\text{in}
\text{SELECT} \text{ROW(f.\varnothing)}, \langle \text{body of } f \text{ recurses to } f^{\text{memo}} \rangle \text{body}
\text{LIMIT} 1
\text{ON CONFLICT} \langle \text{do not insert duplicates into memo} \rangle
\text{RETURNING} \text{val};
\text{\$\$ LANGUAGE} \text{SQL} \text{VOLATILE};$$
```

Figure 24: Wrapping recursive SQL UDF \( f \) to obtain its memoization-based variant \( f^{\text{memo}} \).
Table 1: Call times (in ms) for \( \text{dtw}(i,j) \) with random \( i \in \{1, \ldots, n\} \): memo-wrapped UDF vs. compiled UDF.

<table>
<thead>
<tr>
<th></th>
<th>( \text{dtw}^\text{memo} )</th>
<th></th>
<th>( \text{dtw}^\text{compiled} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>min</strong></td>
<td>7.9</td>
<td><strong>min</strong></td>
<td>3.4</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td>11.5</td>
<td><strong>avg</strong></td>
<td>13.6</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>1,246</td>
<td><strong>max</strong></td>
<td>150</td>
</tr>
</tbody>
</table>

Table 1 reports a comparison of call times for \( \text{dtw}^\text{memo} \) (the wrapped variant of UDF \( \text{dtw} \) of Figure 2) and the compiled UDF. The memoization wrapper indeed delivers the expected profound improvement over the vanilla UDF, avoiding an exponential number of recursive calls. For smaller \( n \), the average call time over 1,000 calls now is comparable with the compiled function. However, \( \text{dtw}^\text{memo} \) still is a recursive SQL UDF and thus exhibits the drawbacks discussed in the introduction: the RDBMS needs to support recursive SQL UDFs in the first place and has to permit recursion to significant depth. Importantly, should \text{memo} not contain an entry for arguments \( \text{args} \) (yet), recursive function calls remain the major cost factor, as documented by the high maximum call times (see columns \text{max} in Table 1).

### 4.3 Linear- and Tail-Recursive UDFs

Linear- and tail-recursive functions [1] exhibit common recursion patterns that allow to notably simplify call graph construction and evaluation. Such functions \( f \) may be characterized by their (prefix tree of) evaluation paths (recall Section 3.4 and Figure 18). Function \( f \) is

- **linear recursive**, if each subtree of paths rooted in a control flow label \( \bigcirc \) contains at most one recursive call site (in the grammar of Figure 11, dark control flow labels \( \bigcirc \) are associated with \text{CASE} expressions).
- **tail-recursive**, if \( f \) is linear recursive and all recursive call sites immediately follow their control flow label.

**Linear recursion.** Because any invocation of a linear-recursive function \( f \) performs at most one recursive call, the resulting call graph will be a chain. Graph traversal thus does not need scheduling: exactly one node will be ready for evaluation in each iteration. We thus can get by with the simplified SQL template for evaluation in Figure 25 which needs no \text{schedule} code. In \text{traverse}, exactly one node \( \text{go} \) will be identified for body evaluation in \text{go} as we walk the call chain back towards its root.

**Tail recursion.** A tail-recursive function \( f \) does not perform any computation after it returns from its (one) recursive tail call [37]. Instead, computation is performed in accumulating function arguments. The accumulators form the final result once the function reaches its base case.

```sql
CREATE FUNCTION \( f(\text{args}) \) RETURNS \( \tau \)
AS $$
\text{WITH RECURSIVE} \quad \text{base_cases}(\text{in}, \text{val}) \quad \text{AS (}
\text{SELECT} \quad \text{go.in} \\
\text{FROM} \quad \text{base_cases} \quad \text{AS} \quad \text{go} \\
\text{WHERE} \quad \text{go.out} = \text{e.in}
\})
\text{AS} \quad \text{go.(in,ret)}$
\text{AS} \quad \text{body} \quad \quad \text{AS}
\text{ITERATE} \quad \text{call_graph}(\text{in}, \text{site}, \text{fanout}, \text{out}, \text{val}) \quad \text{AS} \quad \text{(}
\text{SELECT} \quad \text{b.val} \\
\text{FROM} \quad \text{base_cases} \quad \text{AS} \quad \text{b}
\) \text{RECURSIVE} \text{graph} \quad \text{AS} \quad \text{(}
\text{SELECT} \quad \text{g.out} = \text{e.in} \\
\text{FROM} \quad \text{call_graph} \quad \text{AS} \quad \text{g} \\
\text{WHERE} \quad \text{g.out} = \text{e.in}
\}) \text{AS} \quad \text{go.(in,ret)}$
\})\text{AS} \quad \text{body} \text{ AS}$
$$
```

Since we record the values of these arguments in the call chain’s nodes, the final return value of \( f \) is already known once we have constructed the chain’s base case edge. Graph traversal is not required: we can immediately read the result off the base case edge. A separate evaluation step or SQL template evaluation thus becomes obsolete. The complete SQL template for the compilation of tail-recursive functions \( f \) is reproduced in Figure 26. Note how \text{result} simply extracts the return value \( b\text{.val} \) from table \text{base_cases} as soon as CTE \text{call_graph} has done its job.

**WITH** \text{ITERATE}. Beyond run time savings, tail recursion bears the promise of being space-efficient ("tail recursion needs no stack.") PostgreSQL fails to exploit this potential when it executes non-compiled functional-style UDFs. In fact, PostgreSQL’s recursion depth is limited: even with its maximum stack depth increased to 16 MB (default: 2 MB), PostgreSQL bails out with an overflowing stack after about 18,000 calls.

When we use **WITH** \text{RECURSIVE} to construct the call graph of a function \( f \), we effectively construct a trace of all invocations and their respective arguments. If \( f \) is tail recursive, accumulating this trace is wasted effort: no evaluation step ever revisits the graph and the SQL template of Figure 26 indeed only extracts its single base case edge (see \text{result}). Keeping the most recently generated row in table \text{call_graph}
thus would suffice. This is exactly the behavior of the hypothetical WITH ITERATE construct [28] which has been proposed for inclusion in HyPer [24]. Adding the construct to PostgreSQL 11.3 amounts to a modest local change [11]. If WITH ITERATE replaces WITH RECURSIVE in the template of Figure 26, the system indeed allocates a single-row work table during the entire function evaluation process.

5 PERFORMANCE OF COMPILED UDFs

UDF compilation benefits a wide variety of recursive computations. To make this point, we collected, compiled, and evaluated 10 functions implementing a diversity of algorithmic problems (see Table 2). Use cases include queries over graphs (comp), arithmetic expressions (eval, floyd), expression/program interpretation (eval, vm), string processing (fsm, lcs), traversal of hierarchies (paths, sizes), generation of fractals (mandel), and 2D graphics (march). Each UDF has been realized in the recursive functional style, typically in about 5–15 lines of SQL code. The compiled variants of these functions employ memoization of non-root calls. Linear- and tail-recursive UDFs were compiled using the simplified templates of Section 4.3, the reference counting optimization (Section 4.1) has been applied to all others.

Table 2 lists the average call time in milliseconds across 1,000 random function invocations before and after compilation (columns under Avg. Call Time). For fsm, mandel, and march the individual call times are so low that we count batches of 500, 625, and 9 calls as a single invocation, respectively.

Table 2: A collection of SQL UDFs in functional style and their runtime performance before/after compilation.

<table>
<thead>
<tr>
<th>UDF</th>
<th>Description</th>
<th>Recursion</th>
<th>Avg. Call Time [ms]</th>
<th>Avg. Call Graph Size</th>
<th>Call Times 1,000 invoc.s</th>
<th>Avg. Call Graph Size [memo]</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp</td>
<td>test for connected components in a DAG</td>
<td>2-fold</td>
<td>357 26 (7.2%)</td>
<td>1,000 invoc.s</td>
<td>302 998</td>
<td></td>
</tr>
<tr>
<td>eval</td>
<td>evaluate arithmetic expressions</td>
<td>2-fold</td>
<td>216 20 (9.2%)</td>
<td></td>
<td>1,025</td>
<td></td>
</tr>
<tr>
<td>floyd</td>
<td>find length of shortest path (Floyd-Warshall)</td>
<td>3-fold</td>
<td>&gt;8,000 14 (&lt;1.8%)</td>
<td></td>
<td>1,305 1,302</td>
<td>998</td>
</tr>
<tr>
<td>fsm</td>
<td>parse molecule names using a state machine</td>
<td>linear</td>
<td>659 102 (15.4%)</td>
<td></td>
<td>10 2</td>
<td>195,468</td>
</tr>
<tr>
<td>lcs</td>
<td>find longest common subsequence of strings</td>
<td>2-fold</td>
<td>756 30 (3.9%)</td>
<td></td>
<td>14,865</td>
<td>50,038</td>
</tr>
<tr>
<td>mandel</td>
<td>compute Mandelbrot set</td>
<td>tail</td>
<td>280 27 (9.6%)</td>
<td></td>
<td>13 2</td>
<td>7,701</td>
</tr>
<tr>
<td>march</td>
<td>trace border of 2D object (Marching Squares)</td>
<td>linear</td>
<td>742 28 (3.7%)</td>
<td></td>
<td>391 4</td>
<td>27,018</td>
</tr>
<tr>
<td>paths</td>
<td>reconstruct path names in a file system</td>
<td>tail</td>
<td>474 46 (9.7%)</td>
<td></td>
<td>1,305 1,302</td>
<td>998</td>
</tr>
<tr>
<td>sizes</td>
<td>aggregate file sizes in a directory hierarchy</td>
<td>tail</td>
<td>144 67 (46.5%)</td>
<td></td>
<td>391 269</td>
<td>788</td>
</tr>
<tr>
<td>vm</td>
<td>run a program on a simple virtual machine</td>
<td>tail</td>
<td>207 9 (4.3%)</td>
<td></td>
<td>140 9</td>
<td>50</td>
</tr>
</tbody>
</table>

- The execution of a whole series of function invocations offers opportunities for memoization. The bars in the plots under Avg. Call Graph Size show how the evaluation time of single invocations develops across the series. We see that eval, floyd, or march can effectively reuse prior evaluation efforts while paths and sizes fail to do so.

- The divergence of the call graph sizes for the original and compiled UDFs is another indicator of the memoization potential (columns under Avg. Call Graph Size). Memoization turns entire call subgraphs into base cases and can thus lead to a drastic reduction in the number of calls performed. The price for memoization is the space used by table memo. Column memo reports its size (in rows) after all 1,000 invocations have been performed.

A closer look at some of these UDFs leads to interesting observations about the functions’ behavior at run time:

- **eval**: The two-stage compilation scheme effectively turns the original top-down expression interpreter into a bottom-up variant that processes all independent subexpressions in one iteration step of CTE evaluation. Sharing and memoization potential results from the evaluation of common subexpressions in the input expression DAG.

- **floyd**: Memoization brings the runtime of this textbook-style (yet naive) purely functional implementation of the Floyd-Warshall algorithm down to $O(n^3)$ from $O(3^n)$—in this particular case, it is compilation that enables a practical use of the function in the first place. Table memo will ultimately hold the matrix of shortest path lengths for the input graph. This further brings down call times.

- **fsm**: This deterministic finite state machine sees opportunities for memoization if the parsed molecule names share common suffixes. The runtime impact, however, is limited since the number of recursive calls is small even for the non-compiled UDFs (call graph sizes reflect the lengths of the molecule names of about 10 characters).

---

The PostgreSQL-compatible code for the original and compiled functions is available at [https://github.com/fsUDF/use-cases/](https://github.com/fsUDF/use-cases/).
All four tail-recursive UDFs have been translated using the SQL code template of Figure 26. Use of WITH ITERATE leads to an evaluation in constant (single-row, even) work table space. The presence of varying accumulating arguments, however, limits these functions’ memoization possibilities (see \texttt{mandel}, \texttt{paths}, and \texttt{sizes}).

\textbf{sizes:} The runtime of this function is dominated by a complex array aggregate so that the overhead of recursive calls plays a minor role only. An additional lack of memoization opportunities explains the comparably modest gain in call time performance.

\textbf{vm:} Memoization is possible for this simple virtual machine since we run it on one program (computing Collatz’ $3n + 1$ problem) and only vary the VM’s initial register contents. Compilation lets the UDF interpret 16,665 VM instructions per second (the non-compiled UDF runs at 690 instructions/s).

6 \textbf{MORE RELATED WORK}

A division of complex computation between the database system (where the data lives) and an external programming language (where the processing takes place) is bound to suffer from the DB/PL interface bottleneck. This fundamental problem has been in focus for decades now but remains a hard nut to crack to this day. \texttt{Dbridge}, one notable and long-running research effort initiated by Sudarshan, has developed a variety of techniques to widen the bottleneck—the batching of program-generated queries [31] and the compilation of imperative PL into SQL code, for example [12]. We agree with the colleagues that an answer to the challenge will be found in a combination of techniques developed on both sides of the DB and PL fence. Our take on computation close to the data advocates to leave it in the hands of the RDBMS and to apply PL-inspired compilation techniques [7, 26, 37] directly to SQL, enabling a declarative and readable formulation of recursive functions that can be efficiently evaluated by the SQL processor itself.

In the face of computational workloads generated by machine learning [10, 39], we subscribe to “move the analysis, not the data,” [8, 21] and expect the challenge of complex in-database computation to become ever more pressing.

Recursive functions for SQL, specifically, have been on the table again and again. \texttt{R-SQL} [2] translates SQL functions with self-invocation into a sequence of \texttt{SELECT} and \texttt{INSERT} statements. A generated external program—formulated in Python, for example—then drives the iterative fixpoint evaluation of this SQL core. As a consequence, \texttt{R-SQL} has to repeatedly cross the DB/PL line at query run time, which is exactly what we aim to avoid with our present work.

\texttt{FunSQL} [5] proposes a PL/SQL-like (functional) language in static single assignment form that embeds SQL expressions. The language is compiled into a data-flow graph of algebraic operators—tail recursion is supported and leads to cycles in the graph. We believe that SQL itself should be in focus and try to use it to its fullest, both, as the language in which computation is expressed and as the compilation target.

Recent work on \texttt{RaSQL} represents a pure-SQL approach to recursion [16]. Algorithms are expressed in a generalized recursive CTE form that can be evaluated efficiently, given that the (aggregate functions in the) resulting queries have the \texttt{PreM} property [42]. As we have observed in Section 4.2, memoization is applicable to the arguments of the root call only—the memoization of intermediate calls would need a case-by-case treatment and only add to the complexity and obfuscation of the CTEs.

To reduce the cost of function invocation to zero, \texttt{Froid} [32, 33] describes a conversion of (sufficiently simple) PL/SQL functions into plain SQL code that can then be inlined in the calling query. Although this has not been the focus of this work, functional-style SQL UDFs that have been compiled into recursive CTEs could be inlined as well: no recursive self-invocations remain. (The most recent PostgreSQL 12 has indeed added support for such CTE inlining [30].) Once inlined, the body of the compiled CTE can be optimized and planned together with the enclosing query.

7 \textbf{WRAP-UP}

If an RDBMS implements recursive CTEs, then this work enables that system to efficiently support functional-style UDFs through SQL-to-SQL compilation. This applies to (1) PostgreSQL, where compiled UDFs evaluate significantly faster, (2) SQL Server and Oracle, where compiled UDFs lift stringent restrictions on recursion depth, (3) MySQL and HyPer, which forbid recursion in UDFs in the first place, and also (4) systems like SQLite3, which do not implement UDFs at all (inline the compiled body at the call site to obtain a function-free query).

Recursive computation in SQL arises in user-defined functions but can also occur in the translation of other advanced query constructs. We are developing a compiler [11, 18] that translates PL/SQL code, via an intermediate static-single assignment form, into tail-recursive functions in administrative normal form [9]. In combination with the present results, this enables truly efficient PL/SQL-to-SQL compilation on all RDBMSs mentioned above, including those that have no PL/SQL interpreter in the first place.

We argue that such democratization of efficient recursive and imperative computation in SQL can be a core piece in the in-database programming puzzle.

\textbf{Acknowledgments.} Denis Hirn’s PostgreSQL expertise was invaluable during the development of the SQL code generator. We thank the SIGMOD reviewers for their observations and suggestions that have helped to shape the present paper.
REFERENCES


