Functional Programming on Top of SQL Engines

Tobias Burghardt[0000−0002−9168−9012], Denis Hirn[0000−0001−7040−1780], and Torsten Grust(✉)[0000−0002−8279−0493]

University of Tübingen, Department of Computer Science, Database Reseach Group
Tübingen, Germany
tobias.burghardt@student.uni-tuebingen.de
{denis.hirn,torsten.grust}@uni-tuebingen.de

Abstract. SQL database systems support user-defined functions (UDFs), but they hardly encourage *programming* with these functions. Quite the contrary: the systems' focus on plan-based query evaluation penalizes every function call at runtime, rendering programming with UDFs—especially if these are recursive—largely impractical. We propose to take UDFs for what they are (namely functions) and subject UDFs to a pipeline of function compilation techniques well-established by the FP community (CPS conversion, defunctionalization, and translation into trampolined style, in particular). The result is a non-invasive SQL-level compiler for recursive UDFs that naturally supports memoization and emits iterative CTEs which contemporary SQL engines evaluate efficiently. Functions may not be first class in SQL, but functional programming close to the data can still be efficient.

Keywords: SQL · recursive UDFs · CPS · defunctionalization · trampolined style

1 Recursive SQL UDFs: From 1000s of Plans to One Plan

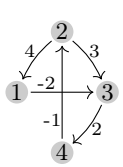
SQL database engines are experts in the plan-based execution of *queries*. Engine internals are specifically designed to support query-to-plan compilation, optimization through plan rewriting, and the—often interpreted—evaluation of the resulting plans.

"If all you have is a hammer, everything looks like a nail." SQL user-defined *functions* (UDFs) receive this plan-centric treatment, too, but in their case the results can only be described as sobering: UDF runtime performance often is disappointing and it is established lore among SQL developers that UDFs are thus best avoided [12, 24, 34]. Indeed, SQL applications pay for the engines' plan-based approach to UDF evaluation literally with every function call.

To make this concrete, consider UDF `floyd(n,s,e)` of Figure 1 which implements Floyd & Warshall's algorithm [16] to find the length of the shortest path between nodes `s` and `e` in a directed graph. The function operates over table `edges` in which a row $\langle h, t, w \rangle$ represents the directed edge $h \xrightarrow{w} t$ of length w (Figure 2a shows a sample graph and its encoding in table `edges`).

```
1 -- length of shortest path (via nodes 1...n) from node s to e
2 CREATE FUNCTION floyd(n int, s int, e int) RETURNS int AS
3 $$
4  SELECT CASE WHEN n = 0 THEN (SELECT edge.w
5                               FROM   edges AS edge
6                               WHERE  (edge.here, edge.there) = (s, e))
7              ELSE LEAST(floyd(n-1, s, e), floyd(n-1, s, n) + floyd(n-1, n, e))
8         END;
9 $$ LANGUAGE SQL STABLE;
```

Fig. 1: Recursive UDF `floyd`, a SQL transcription of function *floyd* of Figure 2b. Yields `NULL` if there is no path from nodes `s` to `e` via nodes `1`...`n`.



edges

here	there	w
1	3	-2
2	1	4
2	3	3
3	4	2
4	2	-1

(a) Graph encoding.

$$floyd(0,s,e) = \begin{cases} w & \text{if } s \xrightarrow{w} e \\ \infty & \text{otherwise} \end{cases}$$
$$floyd(n,s,e) = min\big(floyd(n{-}1,s,e),\; floyd(n{-}1,s,n) + floyd(n{-}1,n,e)\big)$$

(b) Algorithm in its recursive, textbook style.

Fig. 2: Floyd & Warshall's algorithm over a directed graph (no negative cycles). We have $floyd(4,2,3) = 2$ and $floyd(3,3,2) = \infty$, for example.

Note that the code of Figure 1 constitutes a direct transcription of recursive function *floyd* (see Figure 2b) into SQL. This formulation in *functional style* [11] leads to a compact and readable SQL implementation of `floyd(n,s,e)`, yet incurs a flood of recursive UDF calls during evaluation (the UDF of Figure 1 performs $\sum_{i=1}^{n} 3^i = {}^{(3^{n+1}-3)}/_2$ such calls in the absence of memoization). On each top-level or recursive call, the SQL engine creates a new plan context for callee `floyd` to

(P1) compile the `SELECT` block comprising the function's body into a plan,
(P2) improve this initial plan through optimizing plan rewrites,
(P3) instantiate the resulting plan given the current arguments `n`, `s`, and `e`,
(P4) evaluate the plan using a Volcano-style interpreter [19], and finally
(P5) tear down plan data structures before the result is returned and the evaluation of the calling query's plan can resume.

Since the engine needs to keep the plans for callers and callees around, the evaluation of any recursive UDF `f` leads to a nesting of plan contexts $c_0, c_1, \ldots$ as depicted in Figure 3. The repeated effort for plan generation and instantiation (steps P1 and P2, denoted *call* in Figure 3) plus teardown and caller plan resumption (P5, denoted *ret*) adds up to a significant runtime toll which can easily dwarf the productive time spent evaluating the plan for `f`'s body (steps P3 and P4, denoted $\underline{\underline{eval}}$). Plans are rich data structures and, in a sense, the engine finds itself creating and destroying "*super-heavy stack frames*" to drive the evaluation of recursive UDFs.

If we profile the query engine of PostgreSQL (version 13) during the evaluation of a call to `floyd`—which, in this particular case, leads to 88,573 recursive invocations—

we find that the system spends 96% of the overall runtime for function body analysis, query compilation, and plan handling. PostgreSQL implements function inlining (albeit to depth 1 only which thus is of limited use for recursive UDFs) and plan caching: steps P1 and P2 are performed only on the first encounter of a UDF `f` and the resulting plan is saved for reuse during future invocations of `f`. This plan caching, however, does not apply to self-invocations and we observe that PostgreSQL performs steps P1–P5 over and over for every recursive call.

Fig. 3: Nested plan contexts built to evaluate a top-level SQL query `Q` that contains a call to a linear-recursive UDF `f`. Overall evaluation time for `Q` is $t_\omega - t_\alpha$.

The situation certainly is dire, but PostgreSQL indeed fares well if compared to other off-the-shelf SQL DBMSs: MySQL forbids the use of recursion in SQL UDFs (or *stored functions*) in the first place [28, § 25.8], while Oracle and Microsoft SQL Server impose restrictions like recursion-depth limits on UDFs (50 and 32, respectively). PostgreSQL will bail out once the stacked plan contexts exhaust the DBMS server's available process memory [27, 29, 31]. At the bottom line, UDFs appear to be more of an afterthought in SQL engine design than anything else.

Goal: Treating SQL UDFs Like Functions (not Queries). Does the associated runtime penalty thus render the use of function-centric SQL code—and recursion, in particular—impractical? Since UDFs in functional style are one elegant way to express and perform *complex computation close to the data* [11, 36], we would consider this a true loss.

The present work proposes to abstain from immediate (re-)planning on every call and instead take recursive UDFs for what they are: *functions*. This opens up a box that contains tools other than the plan hammer:

(F1) We view a UDF `f` as a plain function f in the sense of functional programming (FP). Function f operates over values of the SQL data model and embeds scalar SQL expressions but otherwise is a vanilla function (Section 2.1).

(F2) To f we then apply a pipeline of established function compilation techniques, see Figure 4. Specifically, we translate f into continuation-passing style (CPS) [2, 38], defunctionalize [35], and finally transform f into trampolined style [17] (Sections 2.2 and 2.3).

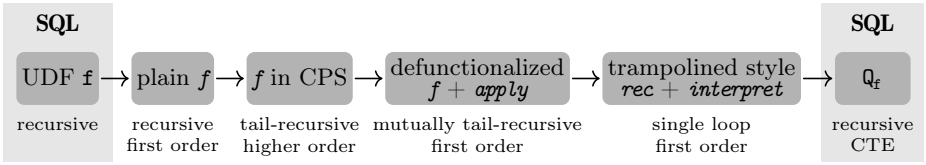


Fig. 4: Using the FP toolbox to compile recursive SQL UDFs into CTEs.

(F3) Function *f* in trampolined style implements a single loop which is readily expressed in terms of a *recursive common table expression* (CTE), *i.e.*, an iterative query form that is widely supported by SQL DBMSs since the advent of the SQL:1999 standard [13, 15, 37]. We obtain SQL query $\texttt{Q}_\texttt{f}$, essentially an CTE-based interpreter loop for UDF `f`.

(F4) $\texttt{Q}_\texttt{f}$ performs no recursive UDF calls and thus will be *planned once* in tandem with its enclosing SQL query. Further, the CTE form provides hooks for a variety of optimizations—memoization, in particular—that render the evaluation of $\texttt{Q}_\texttt{f}$ significantly more efficient than the original UDF `f` which $\texttt{Q}_\texttt{f}$ can replace entirely (Section 3).

The above implements a SQL-level compilation from recursive UDFs to CTEs that is non-invasive and applicable to any DBMS that adheres to SQL:1999—note that this even includes systems that do not natively support recursive UDFs (like MySQL). Section 4 applies this new approach to UDF compilation to a set of recursive functions of varying complexity to demonstrate that function-centric SQL code indeed is one viable way to efficiently compute close to database-resident data.

2 Treating SQL UDFs Like Functions (Not Queries)

The following sketches the SQL-to-SQL compilation of recursive UDFs into CTEs. While we cannot unfold all details, we shine a light on all essential stages of the pipeline in Figure 4.

Boxing SQL subexpressions. We prepare the compilation of UDF `f` by focusing on the essence of the recursive computation that `f` performs, *i.e.*, (1) conditionals that separate base from recursive cases and (2) the sites of recursive calls. These essentials are preserved while all other SQL expressions are wrapped in "black boxes." The contents of these boxes do not affect the subsequent UDF compilation steps and the contained SQL fragments only reappear once the final CTE $\texttt{Q}_\texttt{f}$ is emitted.

Figure 5 shows the boxes $\blacksquare 1, \ldots, \blacksquare 4$ and their contained scalar SQL expressions (in $\lceil\cdot\rfloor$) for UDF `floyd` of Figure 1. Free variables and recursive call sites inside a box $\lceil b \rfloor[e_0, \ldots, e_n]$ are exposed in terms of box parameters e_i: replacing v_i by e_i in b yields the original SQL expression. (We abbreviate both $\lceil v_0 \rfloor[e]$ and $\lceil e \rfloor[\,]$ by e to aid readability.) Besides the boxes, we are left with the top-level `SELECT`

block whose CASE-WHEN-ELSE-END conditional identifies the base and recursive cases in floyd.

```
1  -- length of shortest path (via nodes 0...n) from node s to e
2  CREATE FUNCTION floyd(n int, s int, e int) RETURNS int AS
3  $$                                                              2
4   SELECT CASE WHEN ⌈v0 = 0⌋[n] THEN (⌈SELECT edge.w
5                                 1      FROM   edges AS edge
6                                        WHERE  (edge.here, edge.there) = (v0, v1)⌋[s,e])
7              ELSE ⌈LEAST(v0, v1 + v2)⌋[floyd(⌈v0-1⌋[n], s, e),
8                   3                    floyd(⌈v0-1⌋[n], s, n),
9                                        floyd(⌈v0-1⌋[n], n, e)]
10         END;                                          4×3
11 $$ LANGUAGE SQL STABLE;
```

Fig. 5: UDF floyd and SQL subexpression boxes. (Box 4 occurs three times.)

As discussed here, the compilation scheme applies to recursive SQL UDFs defined via CREATE FUNCTION *f*(*x*1 *τ*1,…,*xn τn*) RETURNS *τ* AS … that adhere to the following syntactic constraints:

1. Return type τ is a scalar SQL type (*i.e.*, *f* may not be a table-valued function), and
2. in a recursive call *f*(*e*1,…,*en*), only the $x_1, \ldots, x_n$ may occur free in the arguments e_i. This restriction ensures that we can lift the call out of its enclosing SQL expression b and place it in the parameter list of box $\lceil b \rfloor$.

2.1 Transition from SQL to FP

Input UDF f is now recast as a first-order function *f* expressed in a simple ML-style language. Importantly, since SQL subexpression boxing has left us with the recursive backbone of the UDF, a (1) CASE-OF conditional, (2) function invocation, and (3) the boxed expressions themselves already make a complete FP target language. In consequence, the atomic types of this language are just the scalar SQL types. For UDF floyd, the resulting function *floyd* is reproduced in Figure 6.

Let us stress once more that this and all following compilation steps leave the boxes ■ intact: in particular, we are never concerned with the FP-equivalent of the rich semantics of SQL's SELECT-FROM-WHERE blocks (as contained in box 2, for example). The boxes are not unpacked before we reach the end of the translation pipeline and are ready to assemble the recursive CTE.

```
1  floyd : (int,int,int) → int
2  floyd(n,s,e) =
3   CASE 1[n] OF
4    true:  2[s,e]
5    false: 3[ floyd(4[n],s,e),
6              floyd(4[n],s,n),
7              floyd(4[n],n,e)]
8
```

Fig. 6: FP-equivalent of UDF floyd. The boxes ■ remain opaque.

2.2 From Recursion Towards Iteration: CPS and Defunctionalization

Since we are heading towards a single-loop interpreter for UDF `f` that does not perform any recursive calls, we proceed by rewriting `f`'s FP-equivalent *f* into *continuation-passing style* (CPS) [1, 38]. *f* in CPS exclusively performs tail calls (which directly translate into iteration later on). Further, CPS explicitly orders the evaluation of function arguments—the CTE-based interpreter, too, will implement just this ordering. ***floyd*** in CPS (Figure 7) computes intermediate results s_1, s_2, s_3 (in this order) and passes these to the continuations A, B, C, respectively.

```
1 floyd : (int,int,int,int → int) → int
2 floyd(n,s,e,k) =
3  CASE 1[n] OF
4   true:  k(2[s, e])
5   false: floyd(4[n],s,e,
6          A{ λs1.floyd(4[n],s,n,
7           B{ λs2.floyd(4[n],n,e,
8            C{ λs3.k(3[s1, s2, s3])))))
```

Fig. 7: *floyd* in CPS. Invocation via *floyd*(n,s,e,λx.x).

Application of the well-established CPS conversion is the first time that we benefit from entering the FP domain. Here, we are free to build on language features like the higher-order continuation arguments `k`, provided that we ensure that such features can be compiled away before we transition back to SQL.

Continuations as data: defunctionalization. In preparation for this back-transition to the SQL domain in which functions are not first class, we opt to represent the continuations in terms of data. See Figure 8 for ***floyd***'s form after this step.

Defunctionalization [35] introduces closure records $\langle \mathtt{k}, env \rangle$ in which tag `k` identifies the continuation (for ***floyd***, $\mathtt{k} \in \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$) and *env* holds the environment of free variables. For ***floyd***, $env \equiv \mathtt{n}, \mathtt{s}, \mathtt{e}, \mathtt{s}_1, \mathtt{s}_2$; we replace variable v by $\square$ if v is undefined in *env* and thus obtain closure records of fixed width ($1 + 5 = 6$ in the case of ***floyd***). The nesting of continuations is encoded in terms of a stack of closure records (see argument `ks` of type `stack` with operations `EMPTY`, `PUSH`, `POP`, `TOP` in Figure 8). Auxiliary function ***apply***(x,ks) inspects tag `k` of the topmost closure record on stack `ks` and invokes the associated continuation on argument `x`. When ***apply*** recognizes continuation tag $\mathtt{k} = \mathtt{Z}$, the final result `x` is returned (see Line 9 in Figure 8). We can thus start the computation via ***floyd***(n,s,e,PUSH(⟨Z, □, □, □, □, □⟩,EMPTY)).

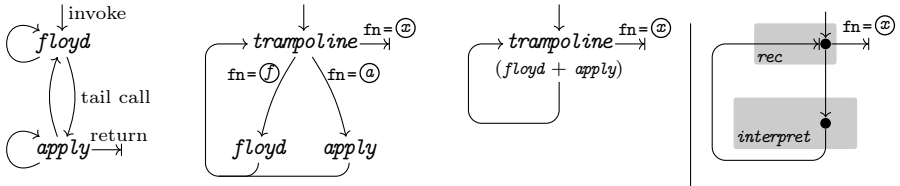
```
1  floyd : (int,int,int,stack) → int
2  floyd(n,s,e,ks) =
3   CASE 1[n] OF
4    true:  apply(2[s, e],ks)
5    false: floyd(4[n],s,e,PUSH(⟨A, n, s, e, □, □⟩,ks))
6
7  apply : (int, stack) → int
8  apply(x,ks) = LET ⟨k, n, s, e, s1, s2⟩ = TOP(ks) IN
9   CASE k OF
10   Z: x
11   A: floyd(4[n],s,n,PUSH(⟨B, n, □, e, x, □⟩,POP(ks)))
12   B: floyd(4[n],n,e,PUSH(⟨C, □, □, □, s1, x⟩,POP(ks)))
13   C: apply(3[s1, s2, x],POP(ks))
```

Fig. 8: *floyd* after defunctionalization. `LET` in Line 7 matches on closure records.

2.3 Trampolined Style: Single Loop Replaces Mutual Recursion

We have arrived at the pair *f*/*apply* of functions which mutually recurse. (Note: an input of n mutually recursive SQL UDFs $\texttt{f}_1, \ldots, \texttt{f}_n$ would lead us to a family $f_1/\cdots/f_n/$*apply* of $n+1$ FP functions at this point.) For the defunctionalized *floyd*, the resulting call graph is depicted in Figure 9a.

The complexity of this call graph is at odds with the single-loop iteration that SQL's recursive common table expressions can express (Section 3 below elaborates on the semantics of recursive CTEs). A better match is offered by *trampolined style* [17] in which a designated *trampoline* function is in charge of dispatching *all* function calls in a given program: to invoke *g* from *f*, (1) *f* tail calls *trampoline*, providing the arguments to be passed on to *g* along with function label `fn` = ⓖ, (2) then *trampoline* invokes *g* as directed by `fn`. *trampoline*'s full control of whether and how the computation proceeds enables a wide variety of applications of trampolined style [17]—here, we are primarily interested in the inherent call graph simplification it provides (see Figure 9b). From here, inlining the bodies of *floyd* and *apply* into *trampoline* yields the single loop we were after (Figure 9c): the evaluation of *trampoline* is iterated until function label argument `fn` = ⓧ directs the program to exit.



(a) After defunctionalization. **(b) Trampolined style (before inlining).** **(c) After inlining of *floyd* and *apply*.** **(d) Single-loop interpreter.**

Fig. 9: Call graphs before and after transformation into trampolined style.

We break the trampolined-style program into two functions: *rec* implements the single-loop iteration and is invariably required to compile any recursive UDF. (Section 3 will show that *rec* embodies SQL's recursive CTE construct.) *rec* iteratively invokes work horse *interpret* which performs the UDF-specific computation, also see Figure 9d.

interpret systematically derives from the *f*/*apply* pair; Figure 10 shows the instance originating from *floyd*/*apply* of Figure 8. After inlining, *interpret* incorporates both functions and the outermost `CASE` of Line 7 inspects label `fn` (of type `fun` = {ⓕ, ⓐ}) to proceed either like *floyd* (ⓕ) or *apply* (ⓐ). By construction, both *floyd* and *apply* exclusively advance computation through mutual invocation (unless they return result `x`). Thus, following trampolined style, in *interpret* we encode a call *floyd*`(n,s,e,ks)` by returning tuple `(`ⓕ`,n,s,e,`□`,ks,`□`)` to *rec*. On the next iteration, *interpret* will proceed like *floyd* as required. Likewise,

```
1  rec : (fun,int,int,int,int,stack,int) → int
2  rec(fn,n,s,e,x,ks,res) = CASE fn OF
3                             ⓧ:  res
4                             ELSE rec(interpret(fn,n,s,e,x,ks,res))

5  interpret : (fun,int,int,int,int,stack,int) → (fun,int,int,int,int,stack,int)
6  interpret(fn,n,s,e,x,ks,res) =
7   CASE fn OF
8    ⓕ: CASE 1[n] OF
9          true:  (ⓐ,□  ,□,□,□,2[s, e]  ,ks                                  ,□)
10         false: (ⓕ,4[n],s ,e ,□       ,PUSH(⟨A,n,s,e,□,□⟩,ks)              ,□)
11   ⓐ: LET ⟨k, n, s, e, s1, s2⟩ = TOP(ks) IN
12       CASE k OF
13         Z:     (ⓧ,□  ,□,□,□,□       ,□                                   ,x)
14         A:     (ⓕ,4[n],s ,n ,□       ,PUSH(⟨B,n,□,e,x,□⟩,POP(ks))         ,□)
15         B:     (ⓕ,4[n],n ,e ,□       ,PUSH(⟨C,□,□,□,s1,x⟩,POP(ks))        ,□)
16         C:     (ⓐ,□  ,□,□,□,3[s1, s2, x],POP(ks)                           ,□)
                  fn  n  s  e    x                  ks                       res
```

Fig. 10: Trampolined-style interpreter (*floyd* and *apply* inlined into *interpret*).

tuple (ⓐ,□,□,□,x,ks,□) encodes a call *apply*(x,ks). Returning (ⓧ,□,□,□,□,□,x) from *interpret* directs *rec* to finish the computation with result x. Given arguments n, s, and e, the evaluation can be started via

$$\mathit{rec}(ⓕ,\mathtt{n},\mathtt{s},\mathtt{e},□,\mathtt{PUSH}(\langle \mathtt{Z},□,□,□,□,□\rangle,\mathtt{EMPTY}),□) \quad . \tag{$*$}$$

Looking closer, *interpret* operates like a UDF-specific *interpreter*:

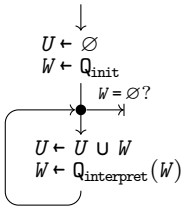
(I1) instructions are of the form (fn,n,s,e,x,ks,res) in which fn and the top continuation k on stack ks determine which action to perform, before

(I2) the next instruction is returned to *rec* to advance (or halt) the computation.

This interpreter consumes and produces tuple-shaped instructions (whose regular format we have tried to indicate via ( ) in Figure 10). We benefit from this regularity when we transcribe the interpreter into its equivalent SQL form in the subsequent section.

3 An Iterative SQL-Based Interpreter for Recursive UDFs

A SQL:1999 *recursive common table expression* [4, 37] takes the syntactic form WITH RECURSIVE *W*(···) AS (Q_{init} UNION ALL $Q_{\text{interpret}}$). It expresses a computation that directly fits the single-loop iteration pattern (and one could argue that *recursive* CTE is a misnomer). The diagram on the left explains this iterative computation, compare it with Figure 9d:



(CTE1) Empty *union table U* which will hold the overall result. Evaluate SQL query Q_{init} and place its rows in *working table W*.

(CTE2) If *W* is empty, return *U* as the final result. Otherwise, add the rows of *W* to *U*.

(CTE3) Evaluate query $Q_{\text{interpret}}$ over the current table *W* and replace the contents of *W* with the resulting rows. Go to CTE2.

```
1  CREATE FUNCTION floyd(n int, s int, e int) RETURNS int AS
2  $$
3   WITH RECURSIVE rec(fn,n,s,e,x,ks,res) AS (
4    SELECT     ⓕ,n      ,s   ,e   ,□              ,PUSH((Z,□,□,□,□,□),EMPTY)          ,□
5      UNION ALL -- recursive union
6    SELECT  interpret.*
7    FROM    rec AS r,
8    LATERAL (SELECT (TOP(r.ks)).*) AS k(k,n,s,e,s1,s2),
10   LATERAL (
14       SELECT ⓐ,□     ,□   ,□   ,2[r.s,r.e]      ,r.ks                              ,□
15       WHERE  r.fn = ⓕ AND 1[r.n]
16         UNION ALL
17       SELECT ⓕ,4[r.n],r.s,r.e,□              ,PUSH((A,r.n,r.s,r.e,□,□),r.ks)     ,□
18       WHERE  r.fn = ⓕ AND NOT 1[r.n]
19         UNION ALL
20       SELECT ⓧ,□     ,□   ,□   ,□              ,□                                 ,r.res
21       WHERE  r.fn = ⓐ AND k.k = Z
22         UNION ALL
23       SELECT ⓕ,4[k.n],k.s,k.n,□              ,PUSH((B,k.n,□,k.e,r.x,□),POP(r.ks)),□
24       WHERE  r.fn = ⓐ AND k.k = A
25         UNION ALL
26       SELECT ⓕ,4[k.n],k.n,k.e,□              ,PUSH((C,□,□,□,k.s1,r.x),POP(r.ks)) ,□
27       WHERE  r.fn = ⓐ AND k.k = B
28         UNION ALL
29       SELECT ⓐ,□     ,□   ,□   ,3[k.s1,k.s2,r.x],POP(r.ks)                         ,□
30       WHERE  r.fn = ⓐ AND k.k = C
31      ) AS interpret(fn,n,s,e,x,ks,res)
32  )           fn   n    s   e         x                    ks                  res
33  SELECT r.res
34  FROM   rec AS r
35  WHERE  r.fn = ⓧ;
36  $$ LANGUAGE SQL STABLE;
```

Fig. 11: Iterative CTE-based interpreter replacing the UDF `floyd` of Figure 1.

We build on these CTEs to construct a SQL formulation of the single-loop UDF interpreter. Figure 11 shows the CTE we obtain from a straightforward transcription of the function pair *rec* + *interpret* of Figure 10 into SQL. (In the SQL code and tables below, □ abbreviates the `NULL` value.) Here, the CTE is wrapped in a SQL UDF `floyd` that could replace the original of Figure 1. The CTE body in Lines 3 to 35, however, can also stand on its own: it contains *no recursive calls* and thus could be inlined at the call sites of `floyd`.

Just like *rec* and *interpret*, the CTE works over tuples `(fn,n,s,e,x,ks,res)`. To kickstart interpretation, $\mathtt{Q}_{\mathtt{init}}$ (*i.e.*, the `SELECT` of Line 4) places an appropriate tuple—or: "instruction", see (∗) above—in working table *W* (named `rec` in Figure 11). The iterated $\mathtt{Q}_{\mathtt{interpret}}$ in Lines 6 to 31 reads the current instruction tuple `r` off table `rec`, processes it, and emits the subsequent instruction that (1) replaces the current tuple in `rec` and (2) also is added to overall result table *U*. Instruction processing entails

(IP1) accessing the topmost continuation on stack `ks`. Much like the `LET` in Line 7 of Figure 10, we use `LATERAL` [37] to bind this continuation to `k` and make it available to the rest of the query. There is a variety of SQL-side implemen-

tation alternatives for stack `ks` and its `PUSH`, `POP`, `TOP` operations. We return to these below.

(IP2) Then, inspection of the function label `r.fn` $\in \{ⓐ, ⓕ, ⓧ\}$ and closure tag `k.k` $\in \{$`A`, `B`, `C`, `Z`$\}$ is used to select the proper subsequent instruction.

Function *interpret* of Figure 10 implements step IP2 in terms of `CASE-OF` multi-way conditionals. Here, we use a tower of predicated `SELECT-WHERE` query blocks chained together via `UNION ALL` (Lines 14 to 30 in Figure 11). Note that the `WHERE` predicates are mutually exclusive such that at most one block can emit an instruction tuple per iteration. In particular, no tuple is produced if `r.fn` = ⓧ: working table `rec` will be empty and the recursive CTE will finish as required (see CTE2 above). Contemporary RDBMSs implement this form of multi-way dispatch efficiently. Consider $\mathtt{Q}_1$ `UNION ALL` $\mathtt{Q}_2$ in which the $\mathtt{Q}_i$ contain `WHERE` predicates p_i that are *independent* of the outcome of their $\mathtt{Q}_i$. On PostgreSQL, this translates into the plan shown on the left. In such a plan, operators `RESULT` evaluate the p_i *before* the sub-plans for the $\mathtt{Q}_i$ are processed [23,31]. Should p_i turn out `false`, the plan for $\mathtt{Q}_i$ is never entered. This exhibits the expected characteristic of multi-way branching and proves to be performant also on other RDBMSs (*e.g.* on Oracle 19c with its `UNION-ALL`/`FILTER` pairs [29]).

```
APPEND
├─RESULT[p1]
│ └⟨plan for Q1⟩
└─RESULT[p2]
  └⟨plan for Q2⟩
```

	fn	n	s	e	x	ks	res
	ⓕ	2	2	3	□	|	□
	ⓕ	1	2	3	□	|	□
	ⓕ	0	2	3	□	|	□
❷	ⓐ	□	□	□	3	$[k_1, \cdot, \cdot]$	□
❸	ⓐ	□	□	□	4	|	□
❹	ⓐ	□	□	□	−2	|	□
❶	ⓐ	□	□	□	2	|	□
	ⓕ	1	2	2	□	|	□
❻	ⓐ	□	□	□	□	|	□
	ⓕ	0	1	3	□	|	□
⓬	ⓐ	□	□	□	−2	|	□
❾	ⓐ	□	□	□	2	$[k_2, \cdot]$	□
❿	ⓐ	□	□	□	2	|	□
	ⓧ	□	□	□	□	|	2

continuation stacks

Fig. 12: CTE union table result for `floyd(2,2,3)`.

The assembly of the instruction tuples themselves directly mimics function *interpret* (*e.g.*, Line 23 of Figure 11 is in correspondence with Line 14 of Figure 10). Once we unfold the contained boxes 1 to 4, we obtain a syntatically complete CTE that can replace the original UDF of Figure 1.

Union table = instruction trace. Given the semantics of the recursive CTE, any invocation of this SQL-based interpreter will yield a union table *U* that collects *a trace of all instructions evaluated by the interpreter*. Each iteration contributes one row to *U*. To illustrate, Figure 12 contains an excerpt of the table resulting from a call `floyd(2,2,3)` (disregard the annotations in ● for now). As expected, we find rows with `fn` = ⓕ which represent the recursive calls to *floyd* (cf. Figure 8), including the top-level call `floyd(2,2,3)`. Rows with `fn` = ⓐ correspond to the application of the current top continuation on stack `ks` to intermediate result `x` (again, recall the invocations of *apply* in Figure 8). The single row with `fn` = ⓧ holds the overall result value in column `res`. Exactly this (grey) table cell is extracted and returned by the final `SELECT` block in Lines 33 to 35 of Figure 11.

3.1 Memoizing the Results of Recursive Calls

A reduction of function call overhead is welcome and Section 4 will assess the performance advantage that the iterative interpreter has over recursive

UDF evaluation. Avoiding the (re-)evaluation of functions altogether, however, certainly beats any execution strategy. This is the promise of *memoization* [6, 26]: once we have spent the effort to evaluate $f(args)$ to value res, memoize the pair $(args, res)$ and immediately respond with res on subsequent calls with arguments $args$. For UDFs like `floyd(n,s,e)` which otherwise performs $O(3^n)$ recursive calls, memoization can be absolutely vital.

The SQL-based interpreter can provide memoization for any UDF `f`. No change to `f` is required. To this end, we associate n-ary UDF `f` with a table `memo(`*args*`,res)` of $n+1$ columns (for `floyd`, this `memo` table has columns |n|s|e|res| with key `(n,s,e)`). The following lines augment `floyd`'s interpreter of Figure 11 to perform a lookup in `memo` for the current arguments `(r.n,r.s,r.e)`:

```
 9 LATERAL ( ⟨lookup in table memo for arguments (r.n,r.s,r.e)⟩ ) AS m("memo?",res),
10 LATERAL (
11     SELECT ⓐ AS fn,r.n,r.s,r.e,m.res AS x,r.ks,□ AS res
12     WHERE  r.fn = ⓕ AND m."memo?"
13       UNION ALL
```

On a successful lookup indicated by `m."memo?" = true`, the memoized value `m.res` is passed directly to the current continuation on top of stack `r.ks` (Line 11). Effectively, the entire subtree of recursive invocations below call `floyd(r.n,r.s,r.e)` is cut short, regardless of whether the call occurs at the top level or deep in the recursion.

How do we populate table `memo`? For one answer, inspect the call tree for top-level invocation *floyd*`(2,2,3)` in Figure 13. When recursive call ❷ to *floyd*`(0,2,3)` has computed intermediate result 3, it passes value `x` = 3 to the top continuation on stack `ks` (which will proceed with call ❸ as determined by CPS). Since union table *U* collects a *log of all such continuation invocations* in rows with `fn` = ⓐ (see column `x` in the row annotated with ❷ in Figure 12), it is a viable source for `memo` entries:

(M1) Run the CTE-based interpreter, obtain union table *U*.

(M2) In *U*, find all rows `u` with `u.fn` = ⓐ. If not already present, insert row `(`*args*`,u.x)` into `memo` where *args* denotes the arguments of the current call.[1]
(We find 13 such rows if the interpreter has evaluated `floyd(2,2,3)`, corresponding to the 13 nodes in the call tree of Figure 13. The lookup in the added Line 9 above will find these entries during subsequent interpreter runs.)

Once completely filled, `floyd`'s `memo` table contains n^3 rows for a graph of n nodes. Builtin index support for the key lookups performed by Line 9 render this form of memoization highly efficient—Section 4 shines a light on this.

Note that the applicability of memoization hinges on UDF `f` being referentially transparent, either generally (`IMMUTABLE` functions [31, § 37.7]) or at least within a transaction context (`STABLE` functions like `floyd` due to its access to table `edges`,

[1] To facilitate M2, we assume that the closure records in `u.ks` additionally provide *args*. In Figure 12, for call ❷, the topmost closure record k_1 would hold the arguments `(n,s,e)` = `(0,2,3)`. Likewise, k_2 would hold `(1,2,3)` for call ❾.

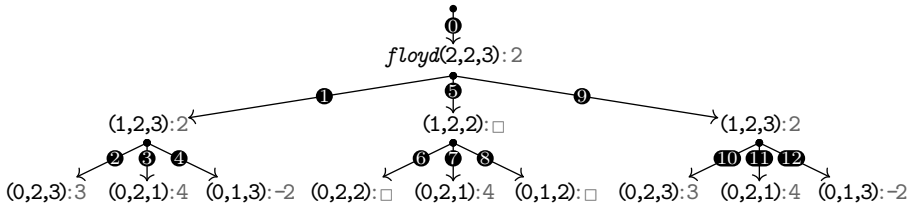


Fig. 13: Call tree for top-level call *floyd*(2,2,3). Edge $\bullet\!\!\xrightarrow{i}$ indicates the ith call performed by the interpreter. Grey values denote the results of the calls.

see Line 9 in Figure 1). These degrees of referential transparency also define the lifetime of table `memo`.

3.2 Optimizations: Slimmer/Shorter Working and Union Tables

The UDF compiler described so far is already fully workable, yet lends itself to a variety of optimizations that help to reduce space usage and runtime. Below, we touch on three improvements that aim to cut down CTE working and union table sizes. The space savings effects increase as we go.

Sharing argument columns. The tuple-shaped instructions reserve separate slots for (1) the arguments of the functions *f* and *apply* (recall Section 2.3) and (2) result value `res`. This defines the width of the rows that we store in the CTE's working and union tables *W* and *U*. The narrower these rows, the less space is needed to hold the instruction log in table *U*.

Since each instruction *either* invokes *f* *or* *apply* *or* returns `res`, these argument tuple slots can be shared between the functions and `res`, provided their types coincide. In the case of UDF `floyd`, *floyd*`(n,s,e,ks)` and *apply*`(x,ks)` currently only share common argument `ks`, leading to a tuple width of $1\,(\texttt{fn}) + 4 + 2 + 1\,(\texttt{res}) - 1\,(\textit{shared}\ \texttt{ks}) = 7$. Given the `int`-typed arguments and `res`, this can be brought down to $1\,(\texttt{fn}) + 1\,(\texttt{n}|\texttt{x}) + 1\,(\texttt{s}|\texttt{res}) + 1\,(\texttt{e}) + 1\,(\texttt{ks}) = 5$. In particular, when multiple mutually recursive UDFs $\texttt{f}_1, \ldots, \texttt{f}_n$ are compiled jointly, argument sharing can drastically reduce the number of `NULL` (□) cells in table *U*.

Continuation stacks outside tables *W* and *U*. The SQL `array` type is one possible SQL-side implementation of the continuation stack in column `ks`. `TOP`, `POP`, and `PUSH` then efficiently operate on the array head element. Still, the array's length is determined by recursion depth—if we recurse deeply, sizable `ks` entries lead to measurable effort when the CTE assembles instruction tuples to be placed in table *W* or appended to *U*.

We have thus experimented with a PostgreSQL extension that hosts the continuation stack *outside* of tables *W* and *U*. Here, column `ks` merely refers to a table-like structure of closure records living in a separate memory region that is private to the SQL query that runs the interpreter. Section 4 reports on the runtime advantages of replacing array-based stacks with this tabular representation.

Avoid building the instruction log. The semantics of `WITH RECURSIVE` entail the construction of union table *U* (see CTE1–CTE3 at the beginning of this section). This instruction log has enabled memoization but, indeed, only the single row with `fn` = ⓧ is essential to return result `res` to the caller. If we opt to forego memoization, we can reach for `WITH ITERATE` [12, 30]. This non-standard variant of SQL's CTE only ever remembers the rows *added last* to working table *W*. No union table *U* is involved at all. Instead, the last non-empty *W* is returned as the final result.

A `WITH ITERATE`-based interpreter only holds the current instruction tuple in memory, the last of which will have `fn` = ⓧ. Should memoization be of no concern for a particular UDF, this optimization promises significant space and runtime savings. We have documented both in the context of earlier related work [12, 22] and also assess the effect of `WITH ITERATE` in Section 4.

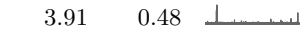
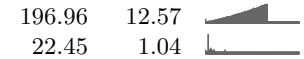
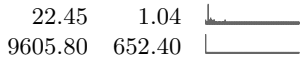
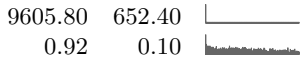
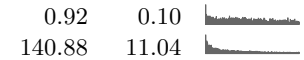
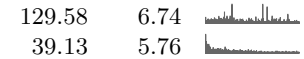
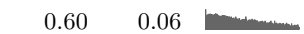
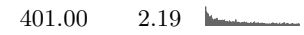
4 Experiments: Functional Programming on Top of PostgreSQL

Recursive UDF processing through the repeated unfolding and planning of function bodies renders relational DBMSs as poor *programming* environments [3, 11]. We argue that it does not have to be this way: the SQL UDF compilation strategy of Figure 4 can turn PostgreSQL into a viable functional programming platform on which complex computation is performed with and *right next to* the tabular data [36].

To make this point, the 10 recursive UDFs of Table 1 address algorithmic problems that would typically not be considered database-resident computations due to (1) their inefficiency when expressed as SQL functions or (2) the forbidding complexity of their manual formulation in terms of a recursive CTE. The UDFs provide implementations of recursive algorithms taken from a variety of domains, ranging from typical database workloads (*e.g.*, over time series or graphs) to more exotic applications. Here, we implement these functions as recursive UDFs in the compact and readable functional style of `floyd` (Figure 1):

- `comps`, like `floyd`, operates over relational adjacency encodings of directed graphs.
- `dtw` stretches (or shrinks) tabular time series to find minimum-distance matchings between two such series, applications of which are found in machine learning or signal processing.
- `eval`, `vm` implement a simple interpreter and virtual machine which enable database applications to regard table-resident data as code.
- `fsm` and `lcs` are representatives of string-based algorithms, a plethora of which are found in the data-intensive bioinformatics domain, for example.
- `paths` realizes a typical bottom-up traversal over hierarchical data (here: a file system directory tree).
- `march` implements *Marching Squares* [25], a classic algorithm in computer graphics that also applies to geographical and map data.
- `mbrot`, finally, constitutes a compute-intensive but data-agnostic algorithm, certainly at the exotic end of the UDF spectrum.

Table 1: Impact of compilation and memoization for 10 recursive SQL UDFs.

UDF	Description	Recursion	Overhead [%]		Time/Call [ms]		Memoize
			UDF	CTE	UDF	CTE	15 000 calls
`comps`	find connected DAG components	2-way	90.64	6.79	3.91	0.48	
`dtw`	Dynamic Time Warping distance	3-way	97.59	1.82	196.96	12.57	
`eval`	evaluate arithmetic expressions	2-way	96.00	3.21	22.45	1.04	
`floyd`	find lengths of shortest paths	3-way	96.74	1.88	9605.80	652.40	
`fsm`	parse with a finite state machine	linear	94.08	15.24	0.92	0.10	
`lcs`	find longest common substring	2-way	98.43	0.67	140.88	11.04	
`mbrot`	compute Mandelbrot set	tail	97.43	29.44	129.58	6.74	
`march`	trace border of 2D object	linear	89.37	3.47	39.13	5.76	
`paths`	construct file system path names	tail	92.27	19.75	0.60	0.06	
`vm`	run program on a virtual machine	tail	98.17	1.61	401.00	2.19	

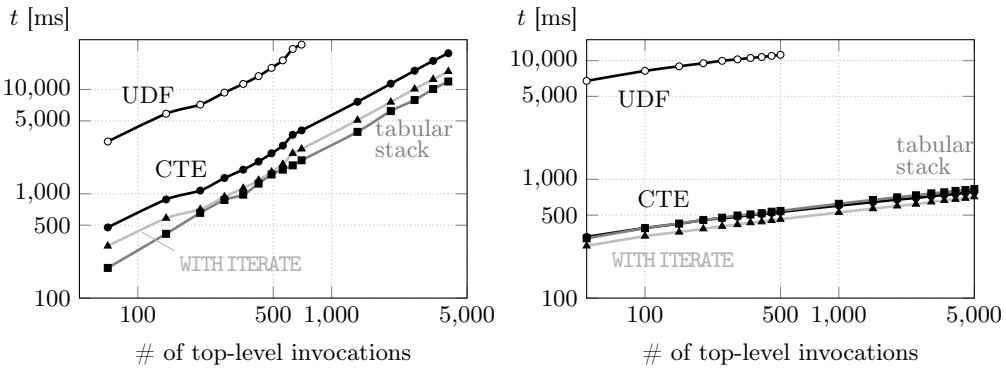
We have also chosen these UDFs to exhibit different recursion patterns (see column **Recursion** in Table 1). Interested readers may evaluate the original as well as compiled UDFs on their local PostgreSQL instances. All required SQL source files are available for download.[2]

For reference: the measurements below report the average of multiple runs, performed on PostgreSQL v13.0. We rely on the vanilla system except where we explicitly mention the use of the query-private table storage extension, recall Section 3.2. The database system was hosted on a 64-bit Linux machine (two AMD EPYC™ 7402 CPUs at 2.8 GHz and 512 GB of RAM, 128 GB of which were assigned to hold the database buffer). The database server's execution stack was set to 6 MB, sufficient to hold the frames of all recursive UDFs in our experiments.

Reducing function call overhead (no memoization). Compilation into CTE form yields iterative SQL queries that do not perform recursive UDF invocations. The saved function call overhead (see Figure 3) is the runtime reduction we are after. Indeed, we find this overhead to account for 95% of the overall runtime of SQL queries `Q` that invoke the UDFs repeatedly with random arguments (averaged across all UDFs, see column **Overhead**). It is now apparent that Figure 3 painted an optimistic picture: the $\underline{\textit{eval}}$ phases of useful work tend to make up no more than $1/20$ of `Q`'s overall timespan $t_\omega - t_\alpha$.

Even without memoization or the optimizations of Section 3.2 enabled, UDF compilation brings this overhead down to an average of about 8%. The remaining overhead is to be attributed to `Q`'s invocation of the non-recursive UDF that wraps the CTE (see Figure 11). If this residual overhead is noticeable—*e.g.*, for computationally lightweight functions like `fsm` and `paths` or frequently-invoked UDFs (`mbrot` is called 16 950 times by `Q`)—it may be advisable to inline the CTE at the UDF call site(s) in `Q`. This significant reduction of the call overhead is reflected by column **Time/Call** which reports on the average runtime per top-level function call before and after compilation. Call time reductions by a factor of 10

[2] `https://github.com/FP-on-Top-of-SQL-Engines/Code`



(a) UDF `march` (Marching Squares). **(b) UDF `eval` (expression evaluation).**

Fig. 14: Runtime of SQL query `Q` before and after UDF compilation, impact of CTE optimizations.

are typical. For UDF `vm`, we even measure an improvement by factor 180: `vm` is structured in terms of 9 conditional branches each of which handles one kind of VM instruction. While the branches are mutually exclusive, all 9 contain recursive calls to `vm` which are unfolded (once) and then planned during each invocation. Post compilation, none of this effort remains.

Impact of memoization. CTEs require time and space to construct union table *U* and our approach to memoization (Section 3.1) aims to exploit this effort. (Aside: tail-recursive functions need no stack and this also applies to their CTE form. Regardless of recursion depth, UDFs `mbrot`, `path`, and `vm` only ever store the initial closure record $\langle$`z`, □, □, □, □, □$\rangle$ of (∗) in column `ks` of tables *W* and *U*, keeping table size and maintenance costs low.) Column **Memoize** of Table 1 documents the runtime impact of memoization once we enable it for a sequence of 15 000 calls to the compiled UDFs. Over time (from left to right), recursive invocations find their random arguments in table `memo` with increasing probability and, as expected, call times go down. Memoization effects are beneficial across all 10 UDFs of Table 1. The behavior of `dtw` reflects our choice of arguments in this particular case: the function is evaluated over time series of increasing length and timings ramp up until the maximum sequence length has been reached—at this point, table `memo` has completely materialized the function [11]. Note that for some UDFs, the effects of memoization only manifest after a larger number of calls: for `comps`, timings develop like ↓‾‾‾‾ over the course of 150 000 invocations.

Zooming in on UDFs `march` and `eval`. In database application contexts, it is typical for a SQL query `Q` to invoke a UDF multiple times. The plots of Figures 14a and 14b report the overall runtime of queries `Q` that perform between 50 and 5 000 top-level invocations of UDFs `march` and `eval`, respectively. Both plots show the order of magnitude runtime differences between UDFs (∾) and their CTE

equivalent (●). The experiment also reveals effects of the CTE optimizations sketched in Section 3.2.

UDF `march` uses linear recursion to implement the *Marching Squares* algorithm that traces the border of an object in the 2D plane [25]. Each step of the recursion adds one point to the border, leading to recursion depths of up to 480 in our experiments (way beyond the depth limits that engines like Oracle or SQL Server enforce). Once `march` is compiled into and evaluated as a CTE, we thus find array-encoded continuation stacks of that same length in column `ks` of tables *W* and *U*. When the CTE-based interpreter pushes onto those stacks and embeds them in the next instruction to execute, PostgreSQL performs costly array copy operations. The tabular continuation stack representation outside *W* and *U* described in Section 3.2 avoids these copy costs and admits contant-time `PUSH` and `POP`. The runtime measurements ■ in Figure 14a manifest these savings. In addition to sizable stacks, `march` has to cope with the construction of a potentially large function result: ever longer arrays of border points accumulate in column `res` of the rows in union table *U*. The switch from `WITH RECURSIVE` to `WITH ITERATE` can avoid the associated row construction and table maintenance effort (at the cost of disabling memoization), see ▲ in Figure 14a.

Both optimizations only show negligble effects for `eval`, however. The UDF performs bottom-up evaluation of subexpressions in a large arithmetic expression tree. Here, the tree depth of 16 defines the maximum recursion depth. This leads to short continuation stacks in column `ks` which are handled efficiently even in their vanilla array representation: the tabular stack optimization does not pay off (■ and ● overlap in Figure 14b). Further, the CTE for `eval` holds comparatively compact results of type `numeric` in column `res` of table *U*. The use of `WITH ITERATE` thus, too, only has marginally impact (▲) and the system fares just fine with the standard `WITH RECURSIVE` construct.

5 More Related Work

The tension between the sobering performance of UDFs and the growing need to move computation closer to high-volume data [7, 9], has led the DB community to double down on its efforts to improve the runtime behavior of procedural SQL code [5, 14, 18, 33].

Recursive UDFs. The massive function call overhead in database engines has prompted earlier work in which we pursued an (arguably more complex) two-phase compilation of recursive SQL UDFs [11]. This approach (1) slices the UDF body to build a call graph (cf. Figure 13) as an explicit tabular data structure, before (2) it uses a recursive CTE to schedule the bottom-up evaluation of that graph. Access to the graph enabled optimizations like the sharing of common subcomputations, but graph construction and maintenance resulted in CTEs that are complex when compared to the simple interpreters emitted by the present CPS-inspired compilation strategy.

R-SQL [3] divides recursive SQL functions into a pure-SQL core and an database-external driver program (*e.g.*, Python code). The latter then controls

the iterative in-database evaluation of the core, requiring repeated crossings of the DB/PL border during query execution. Our SQL-to-SQL compilation scheme exactly aims to avoid any passes through the infamous bottleneck between the database engine and external language processors [32].

We argue that UDFs in functional style lead to compact and idiomatic formulations of in-database computation. *RaSQL* [20] asks developers to express algorithms directly in terms of generalized recursive CTEs that can be evaluated efficiently provided that the resulting queries have the *PreM* property [39]. Recursive CTEs are expressive but their fixpoint semantics [4] and syntactic complexity render them unapproachable for many developers. We would rather bank on a compiler that generates CTEs for us.

Our focus has been on UDFs expressed in SQL, but the chain of compilation steps (starting from "plain f", recall Figure 4) is agnostic about the actual source language. f could be formulated in *Links* [10], for example. In this case, boxes ■ would contain *Links* code which—once translated into SQL using the techniques described by Cheney et al. in [8]—could be placed inside the generated recursive CTE to emit a pure SQL equivalent of recursive *Links* functions (which were not considered to be *shreddable* up to now).

Imperative SQL (PL/SQL, T-SQL). Evaluation of imperative code in PL/SQL procedures (or its PL/pgSQL and T-SQL dialects) involves frequent switches between plan-based query processing and statement-by-statement code interpretation [12]. The resulting friction at runtime motivated work that transforms PL/SQL procedures into pure SQL expressions that can be inlined with the calling SQL query. *Froid* (and its successor *Aggify*) spearheaded research that aims to compile PL/SQL away entirely [21, 22, 34]. Branching off the *Froid* work, we devised a PL/SQL-to-SQL compiler that significantly extends the admissable language constructs, arbitrarily nested iterative control flow, in particular [23]. The compiler emits CTE-based interpreters that resemble those of Section 3 and improves PL/SQL runtime performance significantly.

6 Wrap-Up

We are positive that this SQL UDF compiler is more than a curious ramble through FP land. The runtime savings of about 90% are significant. Applicability is immediate since we pursue a non-invasive, source-level transformation that can be implemented on top of any database engine with SQL:1999 support.

This is work in flux and a variety of knobs remain to be tuned and turned. Among these, we currently study *batching* which evaluates a UDF `f` for a set of n arguments. Batching can be implemented by providing n initial instructions (recall $(*)$), one for each argument. Table `W` will then hold n (not 1) rows during CTE processing such that, effectively, n calls to `f` are evaluated in tandem. Batching will bring down the number of plan context switches between calling query `Q` and `f` once more and also opens up opportunities for parallel function call evaluation.

References

1. Appel, A.: Compiling with Continuations. Cambridge University Press (1992). https://doi.org/10.1017/CBO9780511609619
2. Appel, A.: SSA is Functional Programming. ACM SIGPLAN Notices **33**(4) (1998). https://doi.org/10.1145/278283.278285
3. Aranda, G., Nieva, S., Sáenz-Pérez, F., Sánchez-Hernández, J.: R-SQL: An SQL Database System with Extended Recursion. Electronic Communications of the EASST **64** (Sep 2013)
4. Bancilhon, F.: Naive Evaluation of Recursively Defined Relations. In: On Knowledge Base Management Systems, pp. 165–178. Springer (1986). https://doi.org/10.1007/978-1-4612-4980-1_17
5. Binnig, C., Behrmann, R., Faerber, F., Riewe, R.: FunSQL: It is Time to Make SQL Functional. In: Proc. EDBT/ICDT DanaC. Berlin, Germany (Mar 2012). https://doi.org/10.1145/2320765.2320786
6. Bird, R.: Tabulation Techniques for Recursive Programs. ACM Computing Surveys **12**(4) (Dec 1980). https://doi.org/10.1145/356827.356831
7. Boehm, M., Kumar, A., Yang, J.: Data Management in Machine Learning Systems. Synthesis Lectures on Data Management, Morgan & Claypool (2019). https://doi.org/10.2200/S00895ED1V01Y201901DTM057
8. Cheney, J., Lindley, S., Wadler, P.: Query Shredding: Efficient Relational Evaluation of Queries Over Nested Multisets. In: Proc. SIGMOD (2014). https://doi.org/10.1145/2588555.2612186
9. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J., Welton, C.: MAD Skills: New Analysis Practices for Big Data. Proc. VLDB **2**(2) (Aug 2009). https://doi.org/10.14778/1687553.1687576
10. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web Programming Without Tiers. In: Proc. FMCO. Amsterdam, The Netherlands (2006). https://doi.org/10.1007/978-3-540-74792-5_12
11. Duta, C., Grust, T.: Functional-Style SQL UDFs with a Capital 'F'. In: Proc. SIGMOD (2020). https://doi.org/10.1145/3318464.3389707
12. Duta, C., Hirn, D., Grust, T.: Compiling PL/SQL Away. In: Proc. CIDR (2020)
13. Eisenberg, A., Melton, J.: SQL:1999, Formerly Known as SQL3. ACM SIGMOD Record **28**(1) (Mar 1999). https://doi.org/10.1145/309844.310075
14. Emani, K., Ramachandra, K., Bhattacharya, S., Sudarshan, S.: Extracting Equivalent SQL from Imperative Code in Database Applications. In: Proc. SIGMOD. San Francisco, CA, USA (Jun 2016). https://doi.org/10.1145/2882903.2882926
15. Finkelstein, S., Mattos, N., Mumick, I., Pirahesh, H.: Expressive Recursive Queries in SQL. Joint Technical Committee ISO/IEC JTC 1/SC 21 WG 3, Document X3H2-96-075r1 (Mar 1996)
16. Floyd, R.: Algorithm 97: Shortest Path. Communications of the ACM **5**(6) (1962). https://doi.org/10.1145/367766.368168
17. Ganz, S., Friedman, D., Wand, M.: Trampolined Style. In: Proc. ICFP. Paris, France (Sep 1999). https://doi.org/10.1145/317636.317779
18. Gévay, G., Quiané-Ruiz, J.A., Markl, V.: The Power of Nested Parallelism in Big Data Processing: Hitting Three Flies with One Slap. In: Proc. SIGMOD. Xi'an, Shaanxi, China (Jun 2021). https://doi.org/10.1145/3448016.3457287
19. Graefe, G.: Volcano—An Extensible and Parallel Query Evaluation System. IEEE TKDE **6**(1) (Feb 1994). https://doi.org/10.1109/69.273032

20. Gu, J., Watanabe, Y., Mazza, W., Shkapsky, A., Yang, M., Ding, L., Zaniolo, C.: RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-Aggregate-SQL on Spark. In: Proceedings of the 38th SIGMOD Conference. Amsterdam, The Netherlands (Jun 2019). https://doi.org/10.1145/3299869.3324959
21. Gupta, S., Purandare, S., Ramachandra, K.: Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In: Proc. SIGMOD. Portland, OR, USA (Jun 2020). https://doi.org/10.1145/3318464.3389736
22. Hirn, D., Grust, T.: PL/SQL Without the PL. In: Proc. SIGMOD (2020). https://doi.org/10.1145/3318464.3384678
23. Hirn, D., Grust, T.: One WITH RECURSIVE is Worth Many GOTOs. In: Proc. SIGMOD (2021). https://doi.org/10.1145/3448016.3457272
24. Lawson, C.: How Functions can Wreck Performance. The Oracle Magician **IV**(1) (Jan 2005), http://www.oraclemagician.com/mag/magic9.pdf
25. Maple, C.: Geometric Design and Space Planning Using the Marching Squares and Marching Cube Algorithms. In: Proc. Geometric Modeling and Processing. London, UK (2003). https://doi.org/10.1109/GMAG.2003.1219671
26. Michie, D.: "Memo" Functions and Machine Learning. Nature **218**(306) (Apr 1968). https://doi.org/10.1038/218019a0
27. Microsoft SQL Server 2019 Documentation, http://docs.microsoft.com/en-us/sql
28. MySQL 8.0 Documentation, http://dev.mysql.com/doc/
29. Oracle 19c Documentation, http://docs.oracle.com/
30. Passing, L., Then, M., Hubig, N., Lang, H., Schreier, M., Günnemann, S., Kemper, A., Neumann, T.: SQL- and Operator-Centric Data Analytics in Relational Main-Memory Databases. In: Proc. EDBT. Venice, Italy (2017)
31. PostgreSQL (version 13) Documentation, http://www.postgresql.org/docs/13/
32. Raasveldt, M., Mühleisen, H.: Data Management for Data Science: Towards Embedded Analytics. In: Proc. CIDR (2020)
33. Ramachandra, K., Chavan, M., Guravannavar, R., Sudarshan, S.: Program Transformations for Asynchronous and Batched Query Submission. IEEE TKDE **27**(2) (Feb 2015). https://doi.org/10.1109/TKDE.2014.2334302
34. Ramachandra, K., Park, K., Emani, K., Halverson, A., Galindo-Legaria, C., Cunningham, C.: Froid: Optimization of Imperative Programs in a Relational Database. Proc. VLDB **11**(4) (2018). https://doi.org/10.1145/3186728.3164140
35. Reynolds, J.: Definitional Interpreters for Higher-Order Programming Languages. In: Proc. ACM (1972). https://doi.org/10.1145/800194.805852
36. Rowe, L., Stonebraker, M.: The POSTGRES Data Model. In: Proc. VLDB. Brighton, UK (Sep 1987)
37. SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation, ISO/IEC 9075-2:1999
38. Sussmann, G., Steel, G.: Scheme: An Interpreter for Extended Lambda Calculus. AI Memo (349) (1975)
39. Zaniolo, C., Yang, M., Das, A., Shkapksy, A., Condie, T., Interlandi, M.: Fixpoint Semantics and Optimization of Recursive Datalog Programs with Aggregates. Theory and Practice of Logic Programming **17**(5–6) (Sep 2017). https://doi.org/10.1017/S1471068417000436