Functional-Style SQL UDFs With a Capital 'F'

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

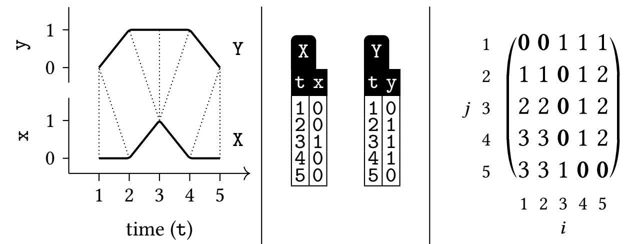
ABSTRACT

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:
Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389707

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]?



(a) Time series X, Y. **(b) Encoding.** **(c) Matrix $dtw(i,j)$.**

Figure 1: Time series $\mathtt{X} = (\mathtt{x}_i)$, $\mathtt{Y} = (\mathtt{y}_j)$, and their tabular encodings. The path of 0s in matrix $dtw(i,j)$ indicates how to warp the series (*e.g.*, $\mathtt{x}_4$ warps to $\mathtt{y}_5$, shown as ⋯ in (a)) for an overall distance of $dtw(5,5) = 0$.

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 $\mathtt{X} = (\mathtt{x}_i)_{i=1\ldots n}$ and $\mathtt{Y} = (\mathtt{y}_j)_{j=1\ldots m}$, 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:

$$\begin{aligned} dtw(0,0) &= 0 \\ dtw(i,0) &= dtw(0,j) = \infty \\ dtw(i,j) &= \left|\mathtt{x}_i - \mathtt{y}_j\right| + \min \begin{Bmatrix} dtw(i-1, j-1) \\ dtw(i-1, j\phantom{{}-1}) \\ dtw(i\phantom{{}-1}, j-1) \end{Bmatrix} . \end{aligned} \tag{DTW}$$

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** `dtw` of Figure 2. The body of UDF `dtw` resembles algorithm DTW as closely as SQL syntax would allow it. In particular,

- the recursion in DTW directly maps to recursive calls of `dtw` (the call sites are marked 1 to 3 in Figure 2), and
- the three-way case distinction in DTW manifests itself as a SQL `CASE` expression in `dtw`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389707

```
CREATE FUNCTION dtw(i int, j int) RETURNS real
AS $$
  SELECT CASE
           WHEN i=0 AND j=0 THEN ④0.0
           WHEN i=0 OR  j=0 THEN ⑤∞  -- 'Infinity'::real
           ELSE (SELECT  abs(X.x - Y.y)
                         +
                          LEAST(①dtw(i-1, j-1),
                                ②dtw(i-1, j  ),
                                ③dtw(i  , j-1))
                 FROM  X, Y
                 WHERE (X.t,Y.t) = (i,j))
         END;
$$ LANGUAGE SQL STABLE STRICT;
```

Figure 2: DTW as a recursive SQL UDF written in functional style. ①, ②, and ③ mark the recursive call sites, ④ and ⑤ designate the non-recursive base cases.

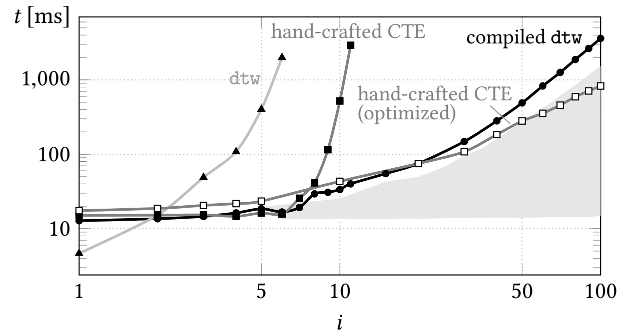


Figure 3: Run time of implementations of dtw(i,i), measured on PostgreSQL 11.3 (logarithmic scales). This work aims for the grey area of low run times.

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 `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].

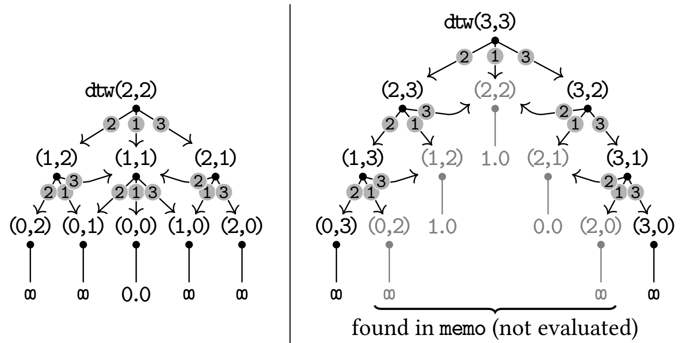
Yet, our recursion-centric style incurs lots of function calls. The naive evaluation of a call `dtw(`i`,`i`)` with $i > 0$ leads to about $0.87 \times ({5.83^i}/\sqrt{i})$ invocations of the function's body [15]. A trace of PostgreSQL 11.3 in fact finds `dtw(2,2)` to plan and run the UDF's body exactly 19 times; `dtw(10,10)` does so 12,146,179 times. Given this, the steeply growing evaluation time of `dtw`, shown as 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. 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 `dtw`, a basic implementation of this compilation strategy can already generate code (see in Figure 3) that surpasses the performance of a purpose-built CTE (see) and rivals that of a specifically optimized variant of the same (). Hand-crafting and tweaking such CTEs tends to lead to complex code that is far from the original algorithm (DTW in this case).

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 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 `dtw(2,2)`.

Sharing. An edge $x \xrightarrow{s} y$ 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 s. 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 `dtw(`i`,`i`)` from $O(5.83^i)$ down to $O(i^2)$. The CTEs generated



(a) Sharing in the call graph for `dtw(2,2)`.

(b) Memoization prunes the call graph for `dtw(3,3)`.

Figure 4: Call graphs for two invocations of UDF `dtw`.

by the UDF compiler build call graphs that exploit sharing opportunities (Section 2 and Section 3.1) for speedy evaluation and the early disposal of obsolete intermediate results (Section 4.1). Implementations of recursive UDFs in comtemporary RDBMSs do not take advantage of call sharing as far as we can tell.

Memoization. If we keep the return values of invocations of f in a table `memo`, we can use these *memoized* results to further trim call graph size. See Figure 4(b) which shows the trimmed call graph of `dtw(3,3)` *after* `dtw(2,2)` and its recursive calls have been evaluated and memoized earlier. Memoized recursive calls act much like base cases, immediately return a value, and thus prune entire call subgraphs. Automatic memoization based on the trimming of an *explicitly* represented call graph is in contrast to programming language implementations of memoization. The latter *dynamically discover* a function's call graph through program or interpreter instrumentation, typically realized in terms of higher-order functions or macro facilities [17, 26]. Since both are unavailable in an RDBMS, we opt for the first-order *call graph as data* implementation (Section 4.2). No changes to the RDBMS or its query evaluator are required.

The generated code operates in two phases in which the construction and trimming of the call graph preceeds the actual function evaluation. We quantify the impact of sharing and memoization when we introduce both in the sequel. If we compile **linear- or tail-recursive functions**, the resulting call graphs are branch-less chains. This presents additional optimization opportunities which we explore in Section 4.3. While we focus on DTW to illustrate our findings, Section 5 will discuss a wide variety of problems that all have compact and efficient formulations as SQL UDFs in the functional, recursion-centric style.

2 FUNCTION CALL GRAPH CONSTRUCTION AND EVALUATION

Let f be a SQL UDF in functional style with scalar return type τ (τ is `real` for our example `dtw`). Recursion is expressed in terms of self-invocation of f at, in general, several call sites (cf. 1 to 3 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.[1] The leaves of g are the non-recursive base cases entered by f (cf. 4 and 5 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*•—*site*→*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 1 to 3. Edges *in*•— *val* towards a leaf *val* indicate that $f(in)$ enters a non-recursive base case that returns result value *val* of type τ.

`call_graph`

in i	in j	site	fan out	out i	out j	val
2	2	1	3	1	1	□
2	2	2	3	1	2	□
2	2	3	3	2	1	□
1	1	1	3	0	0	□
1	1	2	3	0	1	□
1	1	3	3	1	0	□
1	2	1	3	0	1	□
1	2	2	3	0	2	□
1	2	3	3	1	1	□
2	1	1	3	1	0	□
2	1	2	3	1	1	□
2	1	3	3	2	0	□
0	0	□	0	0	0	0.0
0	1	□	0	0	1	∞
1	0	□	0	1	0	∞
0	2	□	0	0	2	∞
2	0	□	0	2	0	∞

Figure 5: Tabular call graph for `dtw(2,2)`.

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*•—*site*→*out* is encoded as row (*in*, *site*, *fanout*, *out*, □) 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*•— *val* maps to row (*in*, □, 0, *in*, *val*). We use □ 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):
                                                       Slices+Calls
calls ← [ ]
FOR EACH call site site of f that would recursively
         invoke f(out) if the arguments are in DO
  calls[site] ← out
edges ← ∅                                              Construct
IF calls ≠ [ ] THEN
  FOR EACH (site, out) IN calls DO
    ADD in •—(site)→ out TO edges
ELSE
  val ← evaluate body of f for arguments in
  ADD in •— val TO graph
edges ← edges \ graph                                  Invoke
FOR EACH · •—(·)→ out IN edges DO
  ADD call_graph(f, out, graph ∪ edges) TO graph
RETURN graph
```

Figure 6: Call graph construction (pseudo code). Invoked via `call_graph`(f, *in*, $\varnothing$), returns a set of edges.

Step 1: Call graph construction can be described as a generic recursive process that accepts a function f and arguments *in*. The pseudo code routine `call_graph`(f, *in*, *graph*) of Figure 6 does exactly that. Parameter *graph*, initially $\varnothing$, 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`:

- `Slices+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.
- `Construct` If we have found that $f(in)$ leads to outgoing calls, construct corresponding edges *in* •—site→ *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.
- `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*, $\varnothing$) 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.

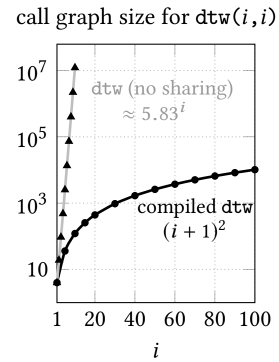


Figure 7: Sharing saves function invocations.

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 `dtw(i,i)`, see •—• 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 `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 `dtw` call graphs of exponential size (▲—▲) which ultimately leads to disastrous function run times.

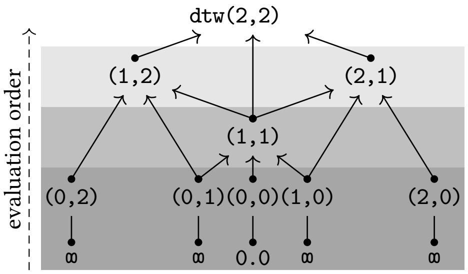
Step 2: Call graph traversal (evaluation). Under our new regime, the generated SQL code finalizes function evaluation via a traversal of f's call graph. Figure 8(a) depicts this traversal for the sample call graph of `dtw(2,2)` shown in Figure 4(a).

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* •—s_1→ out_1 ⋮ —s_n→ out_n, 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 $\{(out_1, val_1), \ldots, (out_n, val_n)\} \subseteq$ `evaluation`. We then evaluate f for argument *in* using a simplified function body in which recursive call site s_i has been replaced by val_i ($i = 1, \ldots, n$). Evaluation of the body will return value *val*, which we enter as row (*in*, *val*) into table `evaluation`.

Figure 8 shows that this iterative evaluation process partitions the call graph for `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 `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



(a) Layer-wise call graph traversal.

evaluation in	val
(2,2)	1.0
(1,2)	1.0
(2,1)	0.0
(1,1)	0.0
(0,2)	∞
(0,1)	∞
(0,0)	0.0
(1,0)	∞
(2,0)	∞

(b) Results.

Figure 8: A bottom-up call graph traversal populates table `evaluation` with the results of all recursive calls performed during the evaluation of `dtw(2,2)`.

```
evaluation(f, args, e, graph):                                Schedule
1  go ← [ ]
2  FOR EACH in•─◦→· IN graph SUCH THAT e[in] = □ DO
3  │ ret ← [ ]
4  │ FOR EACH SUCH call in•─site→out DO
5  │ │ IF e[out] = □ THEN
6  │ │ │ CONTINUE AT 2
7  │ │ ret[site] ← e[out]
8  │ go[in] ← ret
9  returns ← [ ]                                                 Body
10 FOR EACH (in, ret) IN go DO
11 │ val ← evaluate body of f for arguments in
   │        with the nth recursive call site replaced by ret[n]
12 │ returns[in] ← val
13 e ← e ∪ returns                                      Traverse
14 IF e[args] = □ THEN
15 │ RETURN evaluation(f, args, e, graph)
16 RETURN e
```

Figure 9: Call graph traversal (pseudo code).

rows (*in*, *val*) derived from *graph*'s base case edges *in*•— *val* (see above).

Schedule Populate associative array *go* with those unevaluated nodes *in* ($e[in] = \square$) that are now available for evaluation since the return value of all outgoing recursive calls can be found in *e*. In *go*[*in*], store array *ret* which maps f's recursive call sites to their return value recorded in *e*.

Body For each scheduled node *in*, evaluate the body of f in which recursive call sites have been replaced with their return values found in *go*[*in*]. Use *returns* to record the result *val* of the body evaluation for argument *in*.

Traverse Accumulate result rows in *e*. If *graph*'s root *args* is not found in *e* yet, continue the traversal (on the next higher layer). Otherwise, evaluation is complete. Return *e*.

Compiled UDF = call graph construction + traversal. To compile functional-style UDF f, we replace its original body with the composition of call graph construction and traversal. Figure 10 shows the corresponding pseudo code which we will transcribe into proper SQL in the upcoming Section 3.

```
f(args):
graph ← call_graph(f, args, ∅)                        Graph
base_cases ← [ ]                                       Base
FOR EACH in•─ val IN graph DO
│ base_cases[in] ← val
e ← evaluation(f, args, base_cases, graph)             Eval
RETURN e[args]                                         Result
```

Figure 10: (Pseudo) code to replace the original body of UDF f. A SQL formulation is developed in Section 3.

(Note how **Base** primes the result table using *graph*'s base case edges as described above.)

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), and
- 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 τ.

```
udf ::= CREATE FUNCTION f(id, … ,id) RETURNS τ
        AS $$ q $$ LANGUAGE SQL STABLE STRICT;

  q ::= ℓSELECT e AS id, … ,e AS id
        [ ℓFROM   t AS id, … ,t AS id          (optional FROM
        [ ℓWHERE  e ] ]                          and WHERE clauses)

  e ::= ℓ⊗(e, … ,e)                            (n-ary operator ⊗)
      | ℓf(sql, … ,sql)                        (recursive call site)
      | CASE ℓWHEN sql THEN e ELSE e END       (conditional)
      | ℓ(q)                                   (scalar subquery)
      | ℓsql                                   (arbitrary scalar expression)

  t ::= ℓ(q)                                   (tabular subquery)
      | ℓid                                    (table name)

sql ::= any scalar SQL expression without recursive call sites
  f ::= name of recursive SQL UDF to be compiled
 id ::= SQL identifier (table, column, alias, parameter)
  τ ::= scalar SQL type
ℓ, ℓ ::= unique expression labels
```

Figure 11: A grammar for functional-style SQL UDFs. Expression labels (ℓ, ℓ) are internal to the compiler.

The UDF's body is formed by a top-level SELECT-FROM-WHERE block in which scalar and tabular subexpressions (cf. non-terminals e and t) 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 ℓ and ℓ 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 $f(\overline{args})$. In SQL code templates, cursive *type* indicates template text that needs to be replaced. We use $\overline{\text{overlines}}$ to abbreviate comma-separated lists of columns. Further, for $f(\overline{args}) \equiv$ `dtw(i,j)`, $f.\overline{args}$ 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:

Slices Compute two-column table `slices` in which a row (i, out_i) indicates that the evaluation of $f(\overline{args})$ reaches call site s_i and would invoke $f(out_i)$. If call site s_i is not reached for arguments $\overline{args}$, record $(i, \bot)$ in `slices` instead.[2] Table `slices` will carry n rows if f has n recursive call sites (for `dtw`, $n = 3$ with s_i = ⓘ).

[2]Any distinguishable SQL value may be used to represent ⊥ ("bottom").

```
WITH RECURSIVE call_graph(in,site,fanout,out,val) AS (
 SELECT ROW(f.args) AS in, edges.*
 FROM
  (WITH slices(site,out) AS (                                    -- Slices
    SELECT 1 AS site, (slice(f, s1, [f.args])) AS out
      UNION ALL
         ⋮
      UNION ALL
    SELECT n AS site, (slice(f, sn, [f.args])) AS out
   ),
   calls(site,fanout,out,val) AS (
    SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL AS val
    FROM   slices AS s
    WHERE  s.out <> ⊥                                            -- Calls
   )
   TABLE calls
     UNION ALL
   SELECT NULL as site, 0 AS fanout, ROW(f.args) as out,
          (body(f, [NULL::τ,…,NULL::τ], [f.args])) AS val
   WHERE  NOT EXIST (TABLE calls)                               -- Construct
  ) AS edges(site,fanout,out,val)                               -- A

   UNION  -- recursive UNION

 SELECT g.out AS in, edges.*
 FROM   call_graph AS g, LATERAL
  ⟨A with f.args replaced by (g.out).args⟩                       -- Invoke
)
```

Figure 12: Call graph construction (SQL template, compare with Figure 6). Note: code block A occurs twice.

To obtain out_i, we evaluate $slice(f, s_i, [f.\overline{args}])$, 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 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 $slice$(`dtw`, ①, $[i, j]$) is shown in Figure 13.

Calls For each recursive call site s_i that has been reached, collect (the tail of) its call graph edge $\bullet\!\!-\!s_i\!\!\rightarrow out_i$ in table `calls`. We use window aggregate `COUNT(*) OVER ()` [29, §4.2.8] over the non-empty `slices` to find the number of recursive calls performed by $f(\overline{args})$ (recall our discussion of column `fanout` in Section 2).

Construct If, instead, arguments $\overline{args}$ led to a base case (*i.e*, table `calls` is empty), evaluate the body of f for $\overline{args}$ to obtain return value *val*. Construct (the tail of the) base case edge $\bullet\!\!-\!\!-$ *val*. We use $body(f, [e_1, e_2, \ldots], [x_1, x_2, \ldots])$ 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 τ.

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 *out* of the just added graph edges `g` as arguments to f. As per

$slice$(dtw, ①, [i, j]) =

```
(A)SELECT (B)CASE
         (C)WHEN i=0 AND j=0 THEN ⊥
         (D)WHEN i=0 OR  j=0 THEN ⊥
         ELSE (E)((K)SELECT (1)ROW(i-1, j-1)
                 (F)FROM (G)X, (H)Y
                 (I)WHERE (J)(X.t, Y.t) = (i, j))
       END;
```

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

$body$(dtw, [e_1, e_2, e_3], [i, j]) =

```
SELECT CASE
         WHEN i=0 AND j=0 THEN 0.0
         WHEN i=0 OR  j=0 THEN ∞
         ELSE (SELECT  abs(X.x - Y.y)
                       +
                       LEAST((1)e1,
                             (2)e2,
                             (3)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 e_1, e_2, e_3 and i, j, respectively.

the semantics of a recursive CTE using UNION (see Line 22 in Figure 12), construction will terminate once no new graph edges are discovered.

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(in,val), 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.[3]

Body For each such node go, evaluate the body of f with its call sites replaced by the return values found in go.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 $f.\overline{args}$ is indeed found in results.

[3]For brevity, we use custom aggregate array_gather(v,i) which builds array a with $a[i] = v$. Workarounds in standard SQL are easily defined.

```
[WITH RECURSIVE] evaluation(in,val) AS (
 TABLE base_cases
   UNION ALL  -- recursive UNION ALL
  (WITH e(in,val) AS (TABLE evaluation),
   returns(in,val) AS (                                    Body
    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)                              Schedule
   )                                                    Traverse
   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))
 )
)
```

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

```
CREATE FUNCTION f(args) RETURNS τ
AS $$
  WITH RECURSIVE call_graph(in,site,out,val) AS (          Graph
    ⟨see Figure 12⟩
  ),
  base_cases(in,val) AS (          Base
    SELECT g.in, g.val
    FROM   call_graph AS g
    WHERE  g.fanout = 0
  ),
  evaluation(in,val) AS (          Eval
    ⟨see Figure 15⟩
  )
  SELECT e.val                     Result
  FROM   evaluation AS e
  WHERE  e.in = ROW(f.args);
$$ LANGUAGE SQL;
```

Figure 16: Final compiled SQL code to replace the functional-style UDF f. (Compare with Figure 10.)

We note that the code regions **Schedule** and **Traverse** in Figure 15 contain non-monotonic SQL constructs—aggregates and NOT EXISTS, in particular—that some RDBMSs rule out syntactically if they appear in a recursive CTE. The CTEs discussed here are accepted by PostgreSQL [29], HyPer [24], or Umbra [25], for example. Section 4.3 presents simplified evaluation templates for linear- and tail-recursive UDFs that work across all systems that support WITH RECURSIVE.

3.3 Emitted SQL Code

The compiler completes its job by emitting the SQL function f of Figure 16 which glues the two SQL templates together. Just like the pseudo code of Figure 10, **Graph** and **Base** prepare the call graph and base case tables as expected by CTE evaluation (see above). From this table evaluation, **Result**

$$\frac{}{{}^{\ell}f(sql_1,\ldots,sql_n) \bullet\!\!\!\longrightarrow \{[\ell]\}} \qquad \frac{e_i \bullet\!\!\!\longrightarrow \pi_i\big|_{i=1\ldots n}}{{}^{\ell}\otimes(e_1,\ldots,e_n) \bullet\!\!\!\longrightarrow \{[\ell]\} \oplus \bigcup_{i=1\ldots n} \pi_i} \qquad \frac{}{{}^{\ell}sql \bullet\!\!\!\longrightarrow \{[\ell]\}} \qquad \frac{q \bullet\!\!\!\longrightarrow \pi}{{}^{\ell}(q) \bullet\!\!\!\longrightarrow \{[\ell]\} \oplus \pi} \qquad \frac{}{{}^{\ell}id \bullet\!\!\!\longrightarrow \{[\ell]\}}$$

$$\frac{e_1 \bullet\!\!\!\longrightarrow \pi_1 \qquad e_2 \bullet\!\!\!\longrightarrow \pi_2}{\begin{array}{l}\texttt{CASE}\ {}^{\ell}\texttt{WHEN}\ sql\ \texttt{THEN}\ e_1 \\ \quad \texttt{ELSE}\ e_2 \qquad \bullet\!\!\!\longrightarrow \{[\ell]\} \oplus (\pi_1 \cup \pi_2) \\ \texttt{END}\end{array}}$$

$$\frac{e_i \bullet\!\!\!\longrightarrow \pi_{1i}\big|_{i=1\ldots n} \qquad t_j \bullet\!\!\!\longrightarrow \pi_{2j}\big|_{j=1\ldots k} \qquad e \bullet\!\!\!\longrightarrow \pi}{\begin{array}{l}{}^{s}\texttt{SELECT}\ e_1\ \texttt{AS}\ id_{11},\ldots,e_n\ \texttt{AS}\ id_{1n} \\ {}^{f}\texttt{FROM}\quad t_1\ \texttt{AS}\ id_{21},\ldots,t_k\ \texttt{AS}\ id_{2k} \bullet\!\!\!\longrightarrow \{[f]\} \oplus \bigcup_{j=1\ldots k} \pi_{2j} \oplus \{[w]\} \oplus \pi \oplus \{[s]\} \oplus \bigcup_{i=1\ldots n} \pi_{1i} \\ {}^{w}\texttt{WHERE}\quad e\end{array}}$$

Figure 17: $q \bullet\!\!\!\longrightarrow \pi$ derives the set π of evaluation paths for SQL query q. Operator $\oplus$ combines two path sets: $\pi_1 \oplus \pi_2 = \{p_1 \,\|\, p_2 \mid p_1 \in \pi_1, p_2 \in \pi_2\}$ ($\|$ denotes path concatenation).

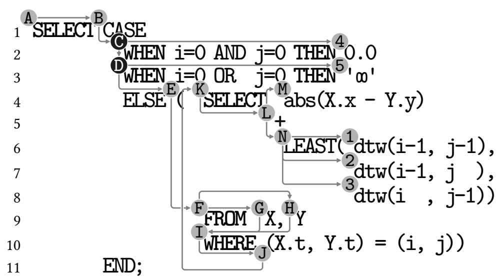


Figure 18: (Prefix tree of) Evaluation paths of UDF `dtw` superimposed on its body. [A, B, ...] shown as A→B→⋯.

extracts f's return value for argument $\overline{args}$ 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 $q \bullet\!\!\!\longrightarrow \{p_1, \ldots, p_n\}$ to compute its set of *evaluation paths*. Each path p_i is a sequence of expression labels $[\ell, \ldots]$ (see Figure 11) which stand in for their associated expressions e_ℓ. Label 1 precedes 2 in p_i (or: $1 <_{p_i} 2$), if e_1 is entered before e_2 in q. Figure 17 defines $\bullet\!\!\!\longrightarrow$ 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 [A, B, C, 4] contains the expressions entered before base case literal `0.0` (label 4) is evaluated. One evaluation path leading to call site 1 is p = [A, B, C, D, E, F, G, I, J, K, L, N, 1]. We find the `FROM` clause (F) and table `X` (G) in p as these bind row variable `X` that is in scope in expression 1. (There is a second path to 1 that routes through table `Y`, label H.) Further, since the `WHERE` clause (I) and its predicate J control whether 1 is evaluated for a given variable binding, we find I $<_p$ 1 and J $<_p$ 1 as well.

Note that evaluation paths reflect dependencies between query subexpression prescribed by the SQL semantics [36] and are independent of any particular order that a RDBMS may choose to evaluate 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 π of evaluation paths via $q \bullet\!\!\!\longrightarrow \pi$. Let $\pi[s_i] \subseteq \pi$ hold the subset of paths that contain call site label s_i.
(2) The labels in $C = \{s_i\} \cup \bigcup_{p \in \pi[s_i]} \{c \mid c <_p s_i\}$ 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 $q^- = q\Downarrow_C$ to remove expressions not relevant to the evaluation of the arguments at call site s_i. $\cdot\Downarrow_C$ is defined in Figure 19 and discussed below.

$$(\textsc{Rec})\quad {}^{\ell}f(sql_1,\dots,sql_n)\Big\Downarrow_C = \begin{cases} {}^{\ell}\texttt{ROW}(sql_1,\dots,sql_n) & \text{if } \ell \in C \\ \bot & \text{otherwise} \end{cases}$$

$$(\textsc{Op})\quad {}^{\ell}\otimes(e_1,\dots,e_n)\Big\Downarrow_C = \begin{cases} e_i\Downarrow_C & \text{if } E = \{i\} \\ \bot & \text{otherwise} \end{cases}\quad \text{with } E = \{i \in 1\dots n \mid e_i\Downarrow_C \neq \bot\}$$

$$(\textsc{Sql})\quad {}^{\ell}sql\Big\Downarrow_C = \begin{cases} {}^{\ell}sql & \text{if } \ell \in C \\ \bot & \text{otherwise} \end{cases}$$

$$(\textsc{Sub})\quad {}^{\ell}(q)\Big\Downarrow_C = {}^{\ell}(q\Downarrow_C)$$

$$(\textsc{Tbl})\quad {}^{\ell}id\Big\Downarrow_C = \begin{cases} {}^{\ell}id & \text{if } \ell \in C \\ \bot & \text{otherwise} \end{cases}$$

$$(\textsc{Case})\quad \begin{array}{l}\texttt{CASE}\ {}^{\ell}\texttt{WHEN}\ sql\ \texttt{THEN}\ e_1 \\ \qquad\texttt{ELSE}\ e_2 \\ \texttt{END}\end{array}\Bigg\Downarrow_C = \begin{array}{l}\texttt{CASE}\ {}^{\ell}\texttt{WHEN}\ sql\ \texttt{THEN}\ e_1\Downarrow_C \\ \qquad\texttt{ELSE}\ e_2\Downarrow_C \\ \texttt{END}\end{array}$$

$$\begin{array}{l} {}^{s}\texttt{SELECT}\ e_1\ \texttt{AS}\ id_{11},\dots,e_n\ \texttt{AS}\ id_{1n} \\ {}^{f}\texttt{FROM}\ \ t_1\ \texttt{AS}\ id_{21},\dots,t_k\ \texttt{AS}\ id_{2k} \\ {}^{w}\texttt{WHERE}\ \ e \end{array}\Bigg\Downarrow_C = \begin{cases} \begin{array}{l} {}^{s}\texttt{SELECT}\ e_i\Downarrow_C\ \texttt{AS}\ id_{1i} \\ {}^{f}\texttt{FROM}\ \ t_1\ \texttt{AS}\ id_{21},\dots,t_k\ \texttt{AS}\ id_{2k} \\ {}^{w}\texttt{WHERE}\ \ e \end{array} & \text{if } \begin{array}{l}\{s,f,w\} \subseteq C, \\ E = \{i\}\end{array} & (\textsc{Select}) \\[2ex] \begin{array}{l} {}^{s}\texttt{SELECT DISTINCT}\ e\Downarrow_C \\ {}^{f}\texttt{FROM}\ \ t_1\ \texttt{AS}\ id_{21},\dots,t_k\ \texttt{AS}\ id_{2k} \end{array} & \text{if } \{f,w\} \subseteq C & (\textsc{Where}) \\[2ex] t_j\Downarrow_C & \text{if } \begin{array}{l} f \in C, \\ T = \{j\}\end{array} & (\textsc{From}) \\[2ex] \begin{array}{l} {}^{s}\texttt{SELECT}\ \bot \\ {}^{f}\texttt{FROM}\ \ t_1\ \texttt{AS}\ id_{21},\dots,t_k\ \texttt{AS}\ id_{2k} \\ {}^{w}\texttt{WHERE}\ \ e \end{array} & \text{otherwise} & (\textsc{Sfw}) \end{cases}$$

$$\text{with } E = \{i \in 1\dots n \mid e_i\Downarrow_C \neq \bot\}\qquad T = \{j \in 1\dots k \mid t_j\Downarrow_C \neq \bot\}$$

Since the labels in C reflect the evaluation order of SELECT-FROM-WHERE blocks (see Figure 17 for the rule for such blocks), $w \in C \Rightarrow f \in C$ and $s \in C \Rightarrow \{f, w\} \subset C$ (*i.e.*, WHERE is evaluated after FROM and SELECT is evaluated after WHERE). Rules Select, Where, From thus cover the possible slicings of the block.

Figure 19: $q\Downarrow_C$ 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 $x_1, x_2, \dots$.

SQL transformation $q\Downarrow_C$ is, again, defined by syntactic case analysis on q. Set C guides the slicing which replaces irrelevant subexpressions by $\bot$, an arbitrary yet distinguished SQL value that SQL template `call_graph` uses to detect that the evaluation of q has *not* entered call site s_i (Figure 12, see **Calls**). Notes on selected cases of $q\Downarrow_C$ in Figure 19:

Rec If this is the call site we are after ($\ell \in C$), 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 ith argument of n-ary operator $\otimes$ ($e_i\Downarrow_C \neq \bot$ 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 $\cdot\Downarrow_C$ replaces irrelevant branches by $\bot$.

Select For a call site found in SELECT expression e_i, keep the FROM and WHERE clauses as these bind in-scope row variables $id_{21}, \dots, id_{2k}$ 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.

Figure 13 shows the result of *slice*(`dtw`, 1, [i,j]). For that example, set C is {A, B, C, D, E, F, G, H, I, J, K, L, N, 1}. Conditional on i and j, the resulting slice either returns (a) the arguments `ROW(i-1, j-1)` of the call at site 1, or (b) $\bot$ if `dtw` enters a base case.

4 COMPILER OPTIMIZATIONS

As Figure 3 indicated for `dtw`, the performance difference between functional-style UDFs () and their compiled counterpart () can already be drastic. Tweaks 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 **Traverse**) to ensure that **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.

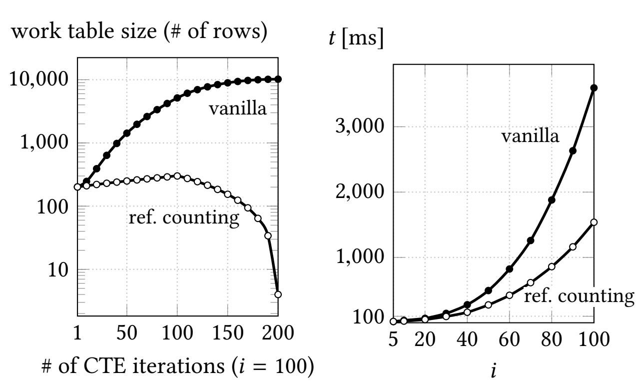
call_g in i | j	ref
(2,2)	□
(2,2)	□
(2,2)	□
(1,1)	3
(1,1)	3
(1,1)	3
(1,2)	1
(1,2)	1
(1,2)	1
(2,1)	1
(2,1)	1
(2,1)	1
(0,0)	1
(0,1)	2
(1,0)	2
(0,2)	1
(2,0)	1

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(a) 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 run time 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).



(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 $\overline{args}$ (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 `IMMUTABLE` 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 $f(\overline{args})$, treat the entries in `memo` like additional *base cases*.
To implement this, use `call_graph`(f, $\overline{args}$, `memo`) to construct the call graph (we used `call_graph`(f, $\overline{args}$, ∅) 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).

memo in	val
(2,2)	1.0
(1,2)	1.0
(2,1)	0.0
(1,1)	0.0
(0,2)	∞
(0,1)	∞
(0,0)	0.0
(1,0)	∞
(2,0)	∞

memo entries serve as call graph leaves

Figure 21: `memo` after `dtw(2,2)`.

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,

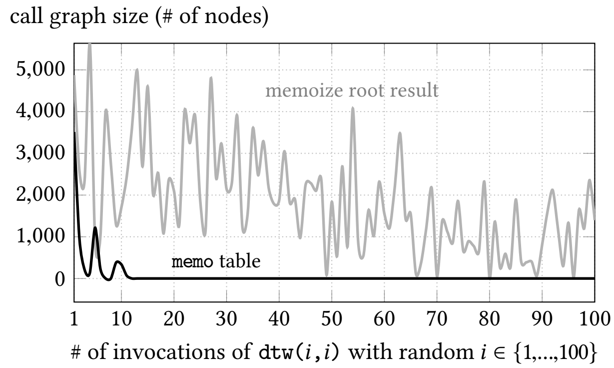


Figure 22: Series of `dtw(i,i)` invocations: Re-using `memo` table entries effectively cuts down call graph size.

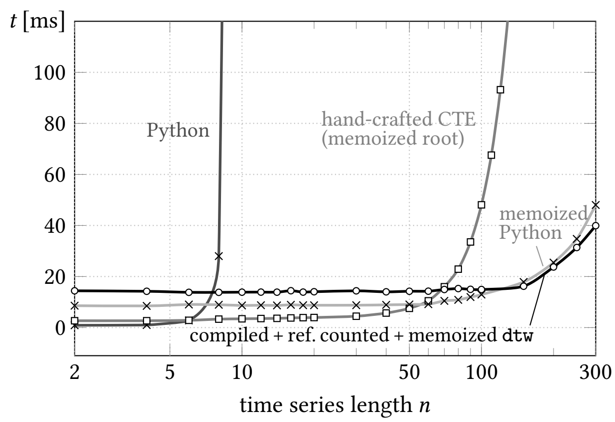


Figure 23: In-database vs. external processing: average evaluation time of `dtw(i,i)` with random $i \in \{1,\ldots,n\}$ (RDBMS: PostgreSQL 11.3, ext. processor: Python 3.7).

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 `dtw(i,i)` with random $i \in \{1,\ldots,100\}$. As expected, memoizing the top-most root call reduces calls graph sizes over time (). The memoization of subgraph roots (), 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 `dtw` () over a hand-crafted CTE (). 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 `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 `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 () 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 ().

Wrapping memoization around recursive UDFs. Memoization does not hinge on compilation. We can obtain f^{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 Memoize extracts f's return value from table `memo` if present. Only if that fails, the original body of f is invoked (Body). It is crucial that body recurses to f^{memo} to ensure that non-root recursive calls, too, are served from `memo`. `INSERT INTO`…`RETURNING val` maintains table `memo` as a side-effect of the evaluation of f^{memo} but ultimately returns `val` as expected by the caller. The function's `VOLATILE` modifier ensures that inserts into `memo` are immediately visible once a recursive call returns [29, §37.7].

```
CREATE FUNCTION f^memo(args) RETURNS τ
AS $$
 INSERT INTO memo(in,val)
   SELECT m.* FROM memo AS m WHERE ROW(f.args) = m.in
  UNION ALL
   SELECT ROW(f.args), ⟨body of f, recurses to f^memo⟩ Body
  LIMIT 1
  ON CONFLICT ⟨do not insert duplicates into memo⟩
  RETURNING val;                                Memoize
$$ LANGUAGE SQL VOLATILE;
```

Figure 24: Wrapping recursive SQL UDF f to obtain its memoization-based variant f^{memo}.

	min	avg	max	min	avg	max	min	avg	max
dtw^{memo}	7.9	**11.5**	1,246	7.9	**12.6**	8,082	7.8	50.1	47,298
compiled	**3.4**	13.6	**150**	**3.4**	14.8	**655**	**3.4**	**23.0**	**15,409**
	50			100			300		

time series length n

Table 1: Call times (in ms) for dtw(i,i) with random $i \in \{1, \dots, n\}$: memo-wrapped UDF *vs.* compiled UDF.

Table 1 reports on a comparison of call times for dtw^{memo} (the wrapped variant of UDF 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, dtw^{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 memo *not* contain an entry for arguments args (yet), recursive function calls remain the major cost factor, as documented by the high maximum call times (see columns **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 ● contains at most one recursive call site (in the grammar of Figure 11, dark control flow labels ● are associated with 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 Schedule code. In Traverse, exactly one node go will be identified for body evaluation in Body 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.

```
[WITH RECURSIVE] evaluation(in,val) AS (
 TABLE base_cases
  UNION
   SELECT go.in,                                          Body
          (body(f, [go.ret,…,go.ret],
                [(go.in).args])) AS val
   FROM   (SELECT g.in, e.val AS ret
           FROM   call_graph AS g, evaluation AS e
           WHERE  g.out = e.in
          ) AS go(in,ret)                                 Traverse
)
```

Figure 25: Call chain traversal for linear-recursive f (replaces the SQL template of Figure 15).

```
CREATE FUNCTION f(args) RETURNS τ
AS $$
 WITH {ITERATE | RECURSIVE} call_graph(in,site,fanout,out,val) AS (
   ⟨see Figure 12⟩
 ),                                                       Graph
 base_cases(in,val) AS (   Base
   ⟨see Figure 16⟩
 )
 SELECT b.val
 FROM   base_cases AS b;   Result
$$ LANGUAGE SQL;
```

Figure 26: Complete SQL template for tail-recursive f. The ITERATE variant allocates a single-row work table.

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 Result simply extracts the return value b.val from table base_cases as soon as CTE call_graph has done its job.

WITH 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 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 Result). Keeping the most recently generated row in table call_graph

UDF	Description	Recursion	Avg. Call Time [ms] UDF	compiled		Call Times 1,000 invoc.s	Avg. Call Graph Size UDF	sharing+memo	|memo|
comps	test for connected components in a DAG	2-fold	357	26	(7.2%)		2,801	254	162,859
eval	evaluate arithmetic expressions	2-fold	216	20	(9.2%)		1,675	13	7757
floyd	find length of shortest path (Floyd-Warshall)	3-fold	>8,000	14	(<1.8‰)		147,621	6	1,329
fsm	parse molecule names using a state machine	linear	659	102	(15.4%)		10	2	195,468
lcs	find longest common subsequence of strings	2-fold	756	30	(3.9%)		14,865	95	50,038
mandel	compute Mandelbrot set	tail	280	27	(9.6%)		13	2	7,701
march	trace border of 2D object (Marching Squares)	linear	742	28	(3.7%)		391	4	27,018
paths	reconstruct path names in a file system	tail	474	46	(9.7%)		1,305	1,302	998
sizes	aggregate file sizes in a directory hierarchy	tail	144	67	(46.5%)		391	269	788
vm	run a program on a simple virtual machine	tail	207	9	(4.3%)		140	9	50

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

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 (`comps`, `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.[4] 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 UDF/compiled 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.

- Compilation leads to a significant reduction in average call time for all UDFs; some functions execute in less than 5% of the time needed by their originals. We address particulars below.
- The execution of a whole series of function invocations offers opportunities for memoization. The bars in the plots under **Call Times** record 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 opportunity 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).

[4] The PostgreSQL-compatible code for the original and compiled functions is available at `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 `mandel`, `paths`, and `sizes`).

`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.

`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 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 places) 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. *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. *R-SQL* [2] translates SQL functions with self-invocation into a sequence of `SELECT` and `INSERT` statements. A generated external program—formulated in Python, for example—then drives the iterative fixpoint evaluation of this SQL core. As a consequence, *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.

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 *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 *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, *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 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.

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

[1] H. Abelson, G.J. Sussman, and J. Sussman. 1996. *Structure and Interpretation of Computer Programs.* The MIT Press.

[2] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández. 2013. R-SQL: An SQL Database System with Extended Recursion. *Electronic Communications of the EASST* 64 (Sept. 2013).

[3] R.W. Barraclough, D. Binkley, S. Danicic, M. Harman, R.M. Hierons, A. Kiss, M. Laurence, and L. Ouarbya. 2010. A Trajectory-Based Strict Semantics for Program Slicing. *Theoretical Computer Science* 411, 11–13 (2010).

[4] D.J. Berndt and J. Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the KDD Workshop.* Seattle, WA, USA.

[5] C. Binnig, R. Behrmann, F. Faerber, and R. Riewe. 2012. FunSQL: It is Time to Make SQL Functional. In *Proceedings of the EDBT/ICDT DanaC Workshop.* Berlin, Germany.

[6] R.S. Bird. 1980. Tabulation Techniques for Recursive Programs. *Comput. Surveys* 12, 4 (Dec. 1980).

[7] R.S. Bird and R. Hinze. 2003. Trouble Shared is Trouble Halved. In *Proceedings of the Haskell Workshop.* Uppsala, Sweden.

[8] M. Boehm, A. Kumar, and J. Yang. 2019. *Data Management in Machine Learning Systems.* Morgan & Claypool.

[9] M. Chakravarty, G. Keller, and P. Zadarnowski. 2004. A Functional Perspective on SSA Optimisation Algorithms. *Electronic Notes in Theoretical Computer Science* 82, 2 (April 2004).

[10] J. Cohen, B. Dolan, M. Dunlap, J.M. Hellerstein, and C. Welton. 2009. MAD Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment* 2, 2 (Aug. 2009).

[11] C. Duta, D. Hirn, and T. Grust. 2020. Compiling PL/SQL Away. In *Proceedings of the 10th CIDR Conference.* Amsterdam, The Netherlands.

[12] K.V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 35th SIGMOD Conference.* San Francisco, CA, USA.

[13] J. Fan, A. Gerald, S. Raj, and J.M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *Proceedings of the 7th CIDR Conference.* Asilomar, CA, USA.

[14] G. Fawaz, H.I. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller. 2019. Deep Learning for Time Series Classification: A Review. *Data Mining and Knowledge Discovery* 33, 4 (July 2019).

[15] M.L. Fredman. 1982. The Complexity of Maintaining an Array and Computing its Partial Sums. *JACM* 29, 1 (Jan. 1982).

[16] J. Gu, Y.H. Watanabe, W.A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. 2019. 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.

[17] P.J. Guo and D. Engler. 2011. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In *Proceedings of the ISSTA Conference.* Toronto, Canada.

[18] D. Hirn and T. Grust. 2020. PL/SQL Without the PL. In *Proceedings of the 39th SIGMOD Conference.* Portland, OR, USA.

[19] E. Horowitz. 1983. *Fundamentals of Programming Languages.* Springer.

[20] P. Hudak, J. Hughes, S. Peyton-Jones, and P. Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the HOPL III Conference.* San Diego, CA, USA.

[21] A. Kumar, M. Boehm, and J. Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 36th SIGMOD Conference.* Chicago, IL, USA.

[22] D. Michie. 1968. "Memo" Functions and Machine Learning. *Nature* 218, 306 (April 1968).

[23] MySQL [n. d.]. *MySQL 8.0 Documentation.* `http://dev.mysql.com/doc/`.

[24] Thomas Neumann. 2011. Efficiently Compiling Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (Aug. 2011).

[25] T. Neumann and M. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the 10th CIDR Conference.* Amsterdam, The Netherlands.

[26] P. Norvig. 1991. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics* 17, 1 (Jan. 1991).

[27] Oracle [n. d.]. *Oracle 19c Documentation.* `http://docs.oracle.com/`.

[28] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. 2017. SQL- and Operator-Centric Data Analytics in Relational Main-Memory Databases. In *Proceedings of the 20th EDBT Conference.* Venice, Italy.

[29] PostgreSQL [n. d.]. *PostgreSQL 11 Documentation.* `http://www.postgresql.org/docs/11/`.

[30] PostgreSQL [n. d.]. *PostgreSQL 12 Documentation.* `http://www.postgresql.org/docs/12/`.

[31] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. 2015. Program Transformations for Asynchronous and Batched Query Submission. *IEEE TKDE* 27, 2 (Feb. 2015).

[32] K. Ramachandra and K. Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019).

[33] K. Ramachandra, K. Park, K.V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. 2018. Froid: Optimization of Imperative Programs in a Relational Database. *Proceedings of the VLDB Endowment* 11, 4 (2018).

[34] L.A. Rowe and M. Stonebraker. 1987. The POSTGRES Data Model. In *Proceedings of the 13th VLDB Conference.* Brighton, UK.

[35] SQL Server [n. d.]. *Microsoft SQL Server 2019 Documentation.* `http://docs.microsoft.com/en-us/sql`.

[36] SQL:1999 [n. d.]. *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation.* ISO/IEC 9075-2:1999.

[37] G.L. Steele Jr. 1977. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the ACM Conference.* Seattle, WA, USA.

[38] F. Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3 (1995).

[39] W. Wang, M. Zhang, G. Chen, H.V. Jagadish, B.C. Ooi, and K.-L. Tan. 2016. Database Meets Deep Learning: Challenges and Opportunities. *ACM SIGMOD Record* 45, 2 (June 2016).

[40] M. Weiser. 1982. Programmer Use Slices When Debugging. *Commun. ACM* 25, 7 (July 1982).

[41] M. Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984).

[42] C. Zaniolo, M. Yang, A. Das, A. Shkapksy, T. Condie, and M. Interlandi. 2017. Fixpoint Semantics and Optimization of Recursive Datalog Programs with Aggregates. *Theory and Practice of Logic Programming* 17, 5–6 (Sept. 2017).