

Viability of Recursive SQL Functions

DISSERTATION
DER MATHEMATISCH-NATURWISSENSCHAFTLICHEN FAKULTÄT
DER EBERHARD KARLS UNIVERSITÄT TÜBINGEN
ZUR ERLANGUNG DES GRADES EINES
DOKTORS DER NATURWISSENSCHAFTEN
(DR. RER. NAT.)

VORGELEGT VON
DIPL.-INFORM. CHRISTIAN DUTA
AUS TÜBINGEN

TÜBINGEN
2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der
Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	05.04.2023
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter/-in:	Prof. Dr. Torsten Grust
2. Berichterstatter/-in:	Prof. Dr. Tilmann Rabl

ABSTRACT

The inclusion of recursive common table expressions (recursive CTEs) in the SQL:1999 standard enabled the developer to implement complex in-database computations, e.g., graph algorithms and others. However, their awkward syntax and rigid fixed-point evaluation method requires highly-specialized expert knowledge in SQL to implement complex computations. This sets the bar quite high for developers to consider implementing such queries.

We propose an alternative way to implement complex in-database computations, which we call *functional-style* SQL UDFs. UDFs implemented in functional-style allow recursive self-invocation inside their own function body. In this publication, we measure the viability of functional-style UDFs from two angles: *readability* and *runtime performance*.

To measure the readability of functional-style UDFs, we conducted a user study in 2020. We presented the participants with tasks from the following topics:

- Choose the correct implementations of the algorithms for the *fibonacci numbers* and *greatest common divisor* formulated in functional-style and recursive CTE.
- Describe and evaluate two unknown UDFs. One is formulated in functional-style and the other as a recursive CTE.
- Implement the *0-1 Knapsack* algorithm based on its textbook style formulation in either functional-style or recursive CTE formulation.

Each participant is then graded based on the score and time needed. In Chapter 2, we present the user study results and compare how well the participants handle these functional-style UDFs compared to recursive CTEs. We discuss if functional-style UDFs can help improve readability for such complex queries.

Besides readability, developers are also interested in the runtime performance of functional-style UDFs. We find that some RDBMSs such as PostgreSQL, Oracle, Microsoft SQL Server, MySQL and SQLite have trouble handling functional-style UDFs

as-is. Performance issues, strict recursion depth limitations, or even outright denying evaluation of functional-style UDFs are the main issues we encountered. In Chapter 4, we describe a SQL-to-SQL compiler which accepts functional-style UDFs, as defined in Chapter 3. The compiler produces standard SQL:1999 recursive CTEs, which replaces the function body of functional-style UDFs so that it no longer exhibits recursive self-involutions. The compiled function body evaluates in a two-phased fashion that

- (1) constructs a call graph top-down and then
- (2) traverses the call graph bottom-up until its root node is reached and the result is returned.

Indeed, the compiler does not rely on intrusive changes to the underlying database engine. Furthermore, this two-phased approach enables optimizations (call sharing, reference counting, linear- and tail-recursion detection, memoization, batching) through small tweaks and improvements to the compiler, which we describe in Chapter 5. We also measure the execution times of various algorithms before and after compilation and discuss the results. We find the runtime performance a developer experiences when compiling functional-style UDFs can improve which is supported by the experiments.

ACKNOWLEDGMENTS

First and foremost, I must thank Prof. Dr. Torsten Grust for granting me the opportunity to research at the unique crossroads between programming languages and database systems at Universität Tübingen. Indeed, his readily available advice, patience and feedback on my journey as a Ph.D. student were invaluable.

I am also profoundly grateful to Dr. Daniyal Kazempour for his advice, feedback and support.

I also thank my colleagues: Daniel O’Grady, Denis Hirn, Tim Fischer, Benjamin Dietrich, and Tobias Müller. For their help, general feedback, and moral support.

Thanks to friends who supported me over the years. Due to the unusual circumstances in recent years, we only kept in touch remotely. I hope this can be remedied in the future when we can finally sit at the same table again.

Lastly, I would be remiss, not to mention my family: Fabienne, my parents, grandparents, siblings, cousins, aunts, and uncles. They kept my motivation high and my spirits strong through emotional support. Thank you.

Contents

1	INTRODUCTION	4
1.1	Functional-Style SQL User-Defined Functions	6
1.2	Performance of Functional-Style SQL UDFs in RDBMSs	6
1.3	Compiling Functional-Style SQL UDFs to Recursive CTEs	9
1.4	More Related Work	14
1.5	Research Focus and Contribution	16
2	RECURSION IN SQL - USER STUDY	17
2.1	Choose the Correct Implementations	18
2.2	Describe a User-Defined Function	20
2.3	Manually Evaluate a User-Defined Function	22
2.4	Implement the 0-1 Knapsack Algorithm	25
2.5	Summary	26
3	GRAMMAR OF FUNCTIONAL-STYLE UDFs	27
4	COMPILING FUNCTIONAL-STYLE UDFs	30
4.1	Step 1: Call Graph Construction	31
4.2	Step 2: Bottom-Up Traversal and Evaluation	34
4.3	SQL Template: Call Graph Construction	36
4.4	SQL Template: Bottom-Up Traversal	39
4.5	Emitting Code	41
4.6	Slicing Functional-Style SQL UDFs	42
5	TWEAKS AND IMPROVEMENTS	50
5.1	Reference Counting	53
5.2	Table-Valued Functions	67

5.3	Linear Recursion	70
5.4	Tail Recursion	73
5.5	Memoization	78
5.6	Batching	86
6	CONCLUSION	92
	APPENDIX A USER STUDY - ONLINE FORM	95
	APPENDIX B COMPLETE TEMPLATES	100
	APPENDIX C SELECTED FUNCTIONAL-STYLE SQL UDFs	107
	C.1 Implementation	107
	C.2 Compilation	111
	C.3 Hand-Crafted Recursive CTEs	120
	APPENDIX D ADDITIONAL USE CASES	124
	D.1 Binomial Coefficient	124
	D.2 Floyd-Warshall Algorithm	124
	D.3 Longest Common Subsequence	125
	D.4 Reachability	126
	D.5 Finite State Machine	126
	D.6 Bounding Box	128
	D.7 Mandelbrot Set Fractals	129
	REFERENCES	131

Preface

Some sentences, phrases, and figures of the following chapters already appeared in or are extensions of previously published works:

CHAPTERS 1, 2 AND 6

[70] Duta, C. (2022). Another Way to Implement Complex Computations: Functional-Style SQL UDF. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '22 New York, NY, USA: Association for Computing Machinery

CHAPTERS 1 AND 3 TO 6

[71] Duta, C. & Grust, T. (2020). Functional-Style SQL UDFs With a Capital 'F'. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20 (pp. 1273–1287). New York, NY, USA: Association for Computing Machinery

The author of this publication contributed to the publication *Compiling PL/SQL Away* [72].

Move your computation close to the data!

L.A. Rowe, M. Stonebraker (1987)

1

INTRODUCTION

MORE AND MORE DEVELOPERS find themselves in a situation where they want computations performed inside a Relational Database Management System (RDBMS). Most popular RDBMSs provide these developers with rich SQL dialects, which come with an extensive toolbox of features to help them implement these computations. But, the more complex these computations become, say recursive algorithms over tabular data, e.g., graph processing or machine learning [73, 64], the deeper the developer must

```
1 WITH RECURSIVE
2 T( $c_1, \dots, c_n$ ) AS (
3    $q_0$ 
4   UNION
5    $q_c(T)$ 
6 )
7 TABLE T;
```

(a) Recursive CTE.

```
1  $u \leftarrow \text{DISTINCT}(q_0)$ 
2  $w \leftarrow u$ 
3 LOOP
4    $i \leftarrow \text{DISTINCT}(q_c(w) \setminus u)$ 
5   IF  $i = \emptyset$  THEN BREAK
6    $u \leftarrow u \cup i$ 
7    $w \leftarrow i$ 
8 END
9 RETURN  $u$ 
```

(b) Pseudo code.

Figure 1.1: The general form of a recursive query T (a) and its evaluation strategy formulated as pseudocode (b). Evaluation requires keeping track of three bag variables: i (*intermediate table*), u (*union table*), and w (*working table*).

```

1 CREATE FUNCTION knap(k int,u int)
2 RETURNS int AS $$
3 CASE
4   WHEN k = 1 THEN 0
5   ELSE (
6     SELECT CASE
7       WHEN i.w > u THEN knap(k-1,u)
8       ELSE
9         GREATEST(knap(k-1,u),
10                  knap(k-1,u-i.w)+ i.p)
11     END
12     FROM items AS i
13     WHERE i.i = k
14   )
15 $$ LANGUAGE SQL STABLE STRICT;

```

(a) Functional-style UDF.

$$\text{knap}(1,u) = 0$$

$$\text{knap}(k,u) = \begin{cases} \text{knap}(k-1,u) & , w_k > u \\ \max\left\{ \text{knap}(k-1,u), \text{knap}(k-1,u-w_k)+p_k \right\} & , \text{otherwise} \end{cases}$$

(b) Textbook-style.

```

CREATE FUNCTION knap(k int,u int) 1
RETURNS int AS $$                2
WITH RECURSIVE sack(i,w,p) AS ( 3
  SELECT 1,0,0                    4
  UNION                            5
  SELECT s.i+1,s.w+c.w,s.p+c.p    6
  FROM sack AS s,items AS i,     7
  LATERAL (                        8
    VALUES (0,0),(i.w,i.p)      9
  ) AS c(w,p)                    10
  WHERE s.w+i.w <= u             11
  AND i.i = s.i+1                12
  AND s.i <= k                    13
  SELECT MAX(s.p) FROM sack AS s; 14
$$ LANGUAGE SQL STABLE STRICT;   15

```

(c) Recursive CTE.

Figure 1.2: Each formulation in (a) – (c) describes the same algorithm to solve the 0-1 Knapsack problem. A rough comparison of the function body in (a) with the body found in (b) shows strong similarities when set side by side. However, compared to (c), the function bodies look almost unrecognizably different.

reach into the SQL toolbox going beyond SELECT-FROM-WHERE clauses. The developer may have to resort to using more expressive language constructs like *recursive common table expression* (recursive CTE) [85], which many popular RDBMSs have adopted since its introduction in SQL:1999 [101].

However, this often requires developers to heavily restructure their computations such that it fits the rigid fixed-point semantics of recursive CTEs. Figure 1.1b outlines how recursive CTEs (in the form described in Figure 1.1a) are evaluated. This is the mold developers have to fit their complex computations into if they need to run such computations inside an RDBMS using standard SQL. Finkelstein et al. argue in favor of the potential benefits recursive CTEs provide through their rigidity [75], but many RDBMS, like PostgreSQL, implement only the restrictions, and none of the benefits [97]. We found that simple syntactic cover-ups lift some of these restrictions, enabling even recursive CTEs that use grouping, aggregates, and anti-joins. So we asked:

“Is there another way to move computation close to the data?”

1.1 FUNCTIONAL-STYLE SQL USER-DEFINED FUNCTIONS

We propose functional-style SQL user-defined functions (we call functional-style UDFs), which give developers another way of expressing complex computations. Essentially, functional-style UDFs are your run-of-the-mill SQL functions that allow recursive self-inocations within their function body.

Consider the *0-1 Knapsack* problem [88], for example. Given items $i \in \{1, \dots, n\}$ where each item has a weight w_i and a value p_i . Function $\text{knap}(n, w)$, recursively defined in its textbook-style formulation in Figure 1.2b, maximizes the sum of values of items that fit into a knapsack of weight w . The recursive CTE formulation (Figure 1.2c) reads almost unrecognizably different from its textbook-style counterpart. The functional-style UDF formulation (Figure 1.2a) fares much better, not significantly altering the textbook-style form. In Chapter 2, we discuss the result of a user study that gauges whether functional-style UDFs are as feasible for developers looking to implement complex computation close to the data. We define the grammar of functional-style UDFs in Chapter 3.

1.2 PERFORMANCE OF FUNCTIONAL-STYLE SQL UDFs IN RDBMSs

It turns out, however, that the native runtime performance of functional-style UDFs in popular RDBMSs is disappointing. We benchmarked five well-known RDBMS to see how well they support functional-style UDFs. SQLite3 does not support SQL UDFs in general [102]. MySQL 8.0’s SQL UDFs do not allow for recursive self-inocations within their function bodies [90]. Their recursive procedures do, albeit with a meager recursion depth of up to 255 set by the `max_sp_recursion_depth` variable [90, §5.1.8]. Microsoft SQL Server 2022 supports recursive self-inocations in SQL UDFs, but only with a hard-coded recursion depth limit [100]. Its `@@NESTLEVEL` variable tracks the recursion depth, and when it exceeds 32, the transaction terminates prematurely. Oracle 19c [95] and PostgreSQL 13 [97] allow for meaningful usage of functional-style UDFs, where the recursion depth can go beyond the 10,000 mark. However, the recursion depth is still limited. Specifically in PostgreSQL 13 we find the recursion depth limited by `max_stack_depth` [97, §19.4.1]. Its value *cannot* exceed the maximum

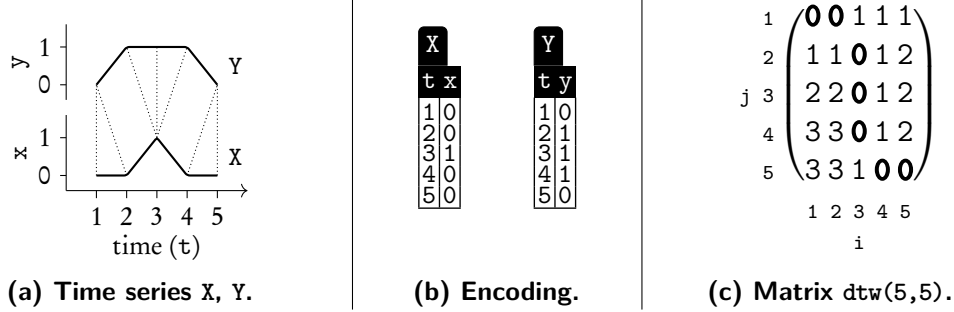


Figure 1.3: Time series $X = x_i$, $Y = y_j$, and their tabular encodings. The path of 0s in matrix $\text{dtw}(5,5)$ indicates how to warp the series (e.g., x_4 warps to y_5 , shown as in (a)) for an overall distance of $\text{dtw}(5,5) = 0$.

stack size set by the operating system. On top of these restrictions, PostgreSQL 13 does not optimize for recursion in functional-style UDFs as far as we can tell. Moreover, the function body is repeatedly parsed and planned for each recursive call due to the absence of plan caching.

Take, for example, *dynamic time warping* (dtw). A widely used time series classification for machine learning [74] and other domains. Given two times series $X = x_i$ and $Y = y_j$ (see Figures 1.3a and 1.3b), where $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$, dtw measures the distance between X and Y if we stretch (or compress) them along the time axis to align both optimally [60]. Its recursive definition in textbook-style reads:

$$\begin{aligned}
 \text{dtw}(0,0) &= 0 \\
 \text{dtw}(i,0) &= \text{dtw}(0,j) = \infty \\
 \text{dtw}(i,j) &= |x_i - y_j| + \min \left\{ \begin{array}{l} \text{dtw}(i-1, j-1) \\ \text{dtw}(i-1, j) \\ \text{dtw}(i, j-1) \end{array} \right\} . \quad (\text{dtw})
 \end{aligned}$$

Figure 1.3 shows how dtw maps ("warps") the series' elements onto each other and measures their overall distance. With tables X and Y of Figure 1.3b in place, developers have two choices of how to implement dtw as a SQL function inside PostgreSQL 13:

- functional-style (Figure 1.4a), which roughly resembles the textbook-style but struggles in terms of runtime performance, or
- as a recursive CTE (Figure 1.4b), which does not resemble the textbook-style but

<pre> 1 CREATE FUNCTION dtw(i int, j int) 2 RETURNS real AS \$\$ 3 CASE 4 WHEN i=0 AND j=0 THEN 0.0 5 WHEN i=0 OR j=0 THEN ∞ -- 'Infinity'::real 6 ELSE (SELECT abs(Z.x - Z.y) 7 + 8 LEAST(①dtw(i-1, j-1), 9 ②dtw(i-1, j), 10 ③dtw(i , j-1)) 11 FROM (X JOIN Y 12 ON ((X.t,Y.t) = (i,j))) AS Z) 13 END; 14 \$\$ LANGUAGE SQL STABLE STRICT; </pre>	<pre> CREATE FUNCTION dtw(i int, j int) RETURNS real AS \$\$ WITH RECURSIVE warp(i,j,val) AS ((SELECT X.t, Y.t, abs(X.x - Y.y) FROM X, Y ORDER BY X.t, Y.t LIMIT 1) UNION SELECT step.i, step.j, MIN(step.val) FROM (SELECT step.i, step.j, warp.val + step.val FROM warp, (VALUES (1,1),(0,1),(1,0)) AS d(i,j), LATERAL (SELECT warp.i+d.i, warp.j+d.j, abs(X.x - Y.y) FROM X, Y WHERE (X.t,Y.t) = (warp.i+d.i,warp.j+d.j)) AS step(i,j,val) WHERE step.i <= dtw.i AND step.j <= dtw.j) AS step(i,j,val) GROUP BY step.i, step.j) SELECT MIN(warp.val) AS dtw FROM warp WHERE (warp.i,warp.j) = (dtw.i,dtw.j); \$\$ LANGUAGE SQL STABLE STRICT; </pre>
<p>(a) Functional-style UDF.</p>	<p>(b) Recursive CTE + optimizations.</p>

Figure 1.4: Functional-style and recursive CTE Implementations of dtw. ①, ②, and ③ in (a) mark the recursive self-inocations (call sites).

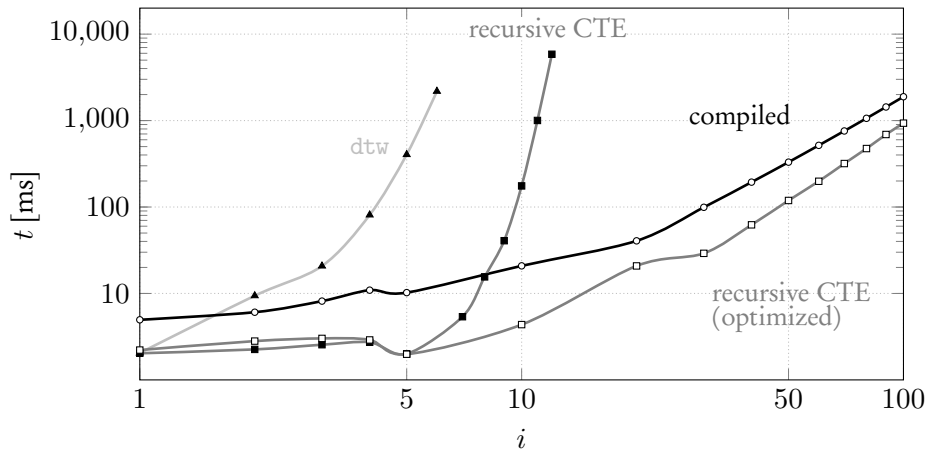


Figure 1.5: Evaluating dtw(i,i): Shows the impact of compilation. Note, how the uncompiled functional-style UDF dtw struggles even when compared to the unoptimized recursive CTE.

allows for intricate optimizations.

In terms of runtime performance, Figure 1.5 reports on the execution times of the functional-style and recursive CTE implementation `dtw` (see 1.4a). These and all following experiments were performed with PostgreSQL 13 running on a 64-bit Linux x86 host with 8 Intel Core™ i7 CPUs clocked at 3.66 GHz and 64 GB of RAM, of which 128 MB were dedicated to the database buffer. Timings were averaged over ten runs, with worst and best runtimes disregarded.

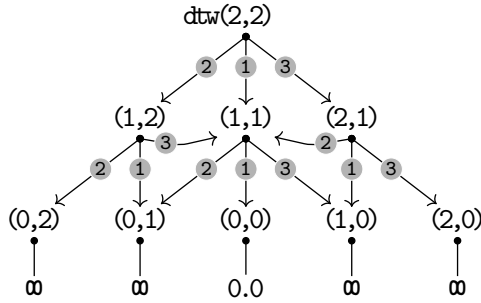
Note how carefully placed optimizations (highlighted in Figure 1.4b) improve performance of the recursive CTE (compare 1.4b with 1.4c in Figure 1.5). Optimizations on this level, however, require a well-developed understanding of recursive CTEs and may be easily overlooked. Thus, we find that, unless we do something about it, neither recursive CTEs nor functional-style UDFs in their natural form satisfy both readability and runtime performance.

1.3 COMPILING FUNCTIONAL-STYLE SQL UDFs TO RECURSIVE CTEs

In this publication, we propose a SQL-to-SQL compiler that

- accepts a functional-style UDF, say f (for example `dtw`, see Figure 1.4a),
- compiles the function body of f into semantically equivalent recursive CTEs which do not require any recursive self-involutions, and
- promises this without invasive modifications to the underlying RDBMS.

Once compiled, f is entirely replaced by its compiled counterpart, which makes it also feasible for systems that do not natively allow for such UDFs in functional-style. The compiled function body of f may even be inlined into the SQL query that invokes f , which enables functional-style UDFs for systems that lack any UDF support. In Chapter 4, we describe the SQL-to-SQL compiler in detail. Using function compilation is only the first step for functional-style UDFs to free developers from the need to come up with hand-crafted CTEs.




call_graph					
in		site	fanout	out	val
i	j		i	j	
(2,2)		□	□	(2,2)	□
(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)	∞

(a) The call graph describes edges representing either a recursive call $in \bullet \text{site} \rightarrow out$ or a non-recursive base case $in \bullet val$.

(b) Tabular representation of the call graph in (a). We use □ to abbreviate SQL's NULL.

Figure 1.6: The call graph for invocation of dtw(2,2).

The ultimate goal is to tweak and improve the compiler such that the execution times of the compiled UDFs approach that of its hand-crafted recursive CTE counterparts. Compilation enables developers to utilize optimizations such as sharing, linear- and tail-recursion optimization, memoization, and batching (Chapter 5). These optimizations are independently applicable and rely on an explicit tabular representation of the UDF's *call graph* which the compiled function f internally maintains. Figure 1.5 (specifically ) gives a preview of what to expect when compiling dtw with optimizations enabled.

SHARING

Figure 1.6a shows the call graph of dtw(2,2). An edge $x \bullet \text{site} \rightarrow y$ in the call graph of a recursive function f indicates that the evaluation of $f(x)$ has led to a recursive call $f(y)$ at call site site . Recursive calls that are shared by multiple invocations of f —like (1,1), (0,1), and (1,0) in Figure 1.6a—indicate the potential to avoid repeated compu-

tations. The naive evaluation of $\text{dtw}(i, i)$ with $i > 0$ leads to about $0.87 \times (5.83^i / \sqrt{i})$ recursive calls [77]. Enabling **sharing** collapses all recursive calls shared by multiple invocations into one node, which can drastically reduce the call graph size and thus evaluation effort (see Figure 1.6b for its tabular representation). In the case of $\text{dtw}(i, i)$ from $O(5.83^i)$ down to $O(i^2)$. Indeed, the recursive CTEs (generated by the function compiler) build such call graphs that exploit sharing opportunities for increased evaluation speed (Section 4.1.2). Contemporary RDBMS do not take advantage of call sharing when evaluating recursive UDFs, as far as we can tell.

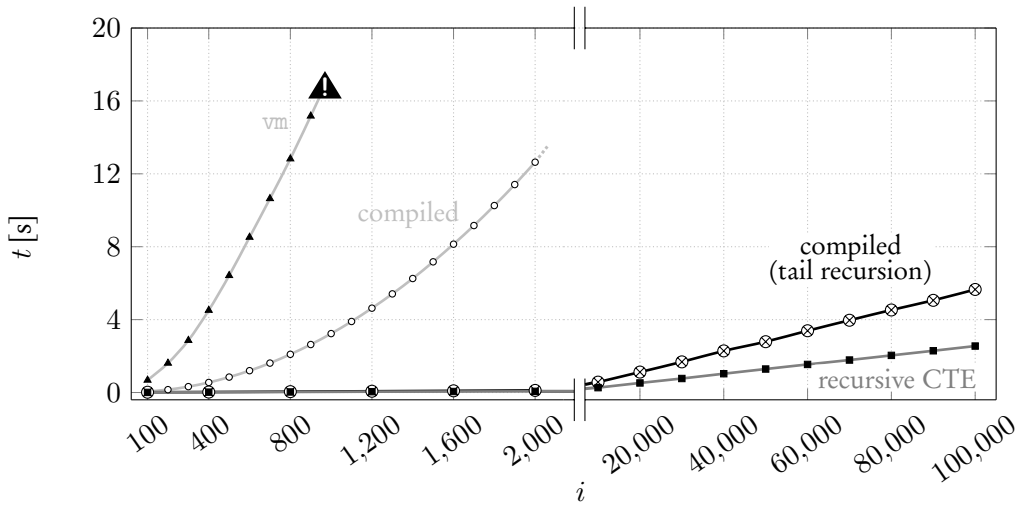


Figure 1.7: Evaluating $\text{vm}(I_0, R)$ where R is the initial register state which is set to repeat a loop i times. With increasing i , evaluation of vm before compilation terminates prematurely due to recursion depth limitation (▲).

LINEAR- AND TAIL-RECURSION

Evaluating **linear- or tail-recursive functions** produces call graphs that are branch-less chains. This assumption allows the compiler to utilize additional optimization opportunities, which we explore in Sections 5.3 and 5.4. Take, for example, program interpretation implemented in functional-style: $\text{vm}(I_p, R)$ takes a program instruction I_p at position $p > 0$ and an initial register state R and continues evaluating instructions until a `hlt`-instruction is reached and a register value is returned. Single registers in R at position k can be accessed with $R[k]$. Binary operator $R[t] \leftarrow e$ replaces the value of

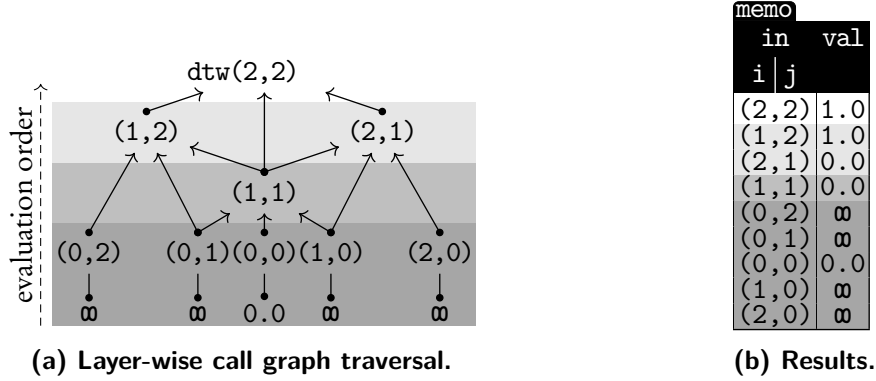
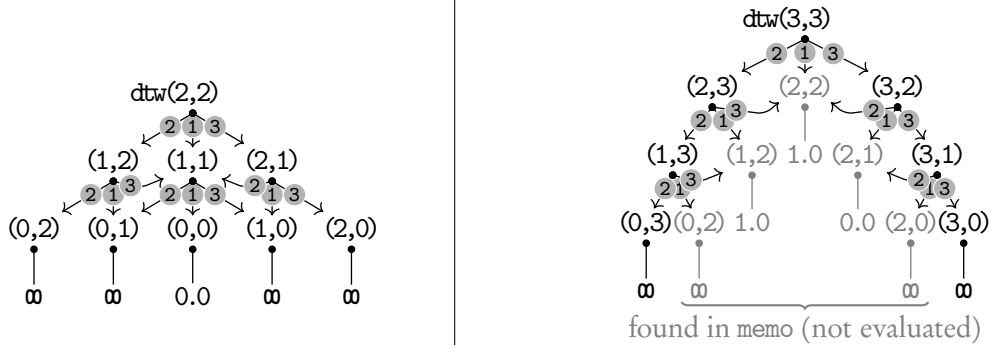


Figure 1.8: A bottom-up call graph traversal populates table `memo` with the results of all recursive calls performed during the evaluation of `dtw(2,2)`.

a single register `t` in `R` with the result of expression `e` and returns the modified registers. Function `vm` is recursively defined as:

$$\begin{aligned}
\text{vm}((i, \text{hlt}, s), R) &= R[s] \\
\text{vm}((i, \text{lod}, t, x), R) &= \text{vm}(I_{i+1}, R[t] \leftarrow x) \\
\text{vm}((i, \text{mov}, t, s), R) &= \text{vm}(I_{i+1}, R[t] \leftarrow R[s]) \\
\text{vm}((i, \text{add}, t, s_1, s_2), R) &= \text{vm}(I_{i+1}, R[t] \leftarrow R[s_1] + R[s_2]) \\
\text{vm}((i, \text{mul}, t, s_1, s_2), R) &= \text{vm}(I_{i+1}, R[t] \leftarrow R[s_1] * R[s_2]) \\
\text{vm}((i, \text{div}, t, s_1, s_2), R) &= \text{vm}(I_{i+1}, R[t] \leftarrow R[s_1] / R[s_2]) \\
\text{vm}((i, \text{mod}, t, s_1, s_2), R) &= \text{vm}(I_{i+1}, R[t] \leftarrow R[s_1] \% R[s_2]) \\
\text{vm}((i, \text{jmp}, a), R) &= \text{vm}(I_a, R) \\
\text{vm}((i, \text{jeq}, t, s, a), R) &= \begin{cases} \text{vm}(I_a, R) & , R[s] = R[t] \\ \text{vm}(I_{i+1}, R) & , \text{otherwise.} \end{cases}
\end{aligned} \tag{vm}$$

Function `vm` is tail-recursive, as each branch recursively calls `vm` directly. Figure 1.7 reports that running `vm` precompilation terminates prematurely whenever evaluating reaches roughly the 1,000th instruction due PostgreSQL’s limited recursion depth [97]. Compiling this function with tail-recursion detection disabled (`compiled`) is not bound by recursion depth, but performance is disappointing compared to the hand-crafted recursive CTE. With tail recursion detection, execution times close in on the performance of the hand-crafted recursive CTE.



(a) Sharing in the call graph for `dtw(2,2)`. (b) Memoization prunes the call graph for `dtw(3,3)`.

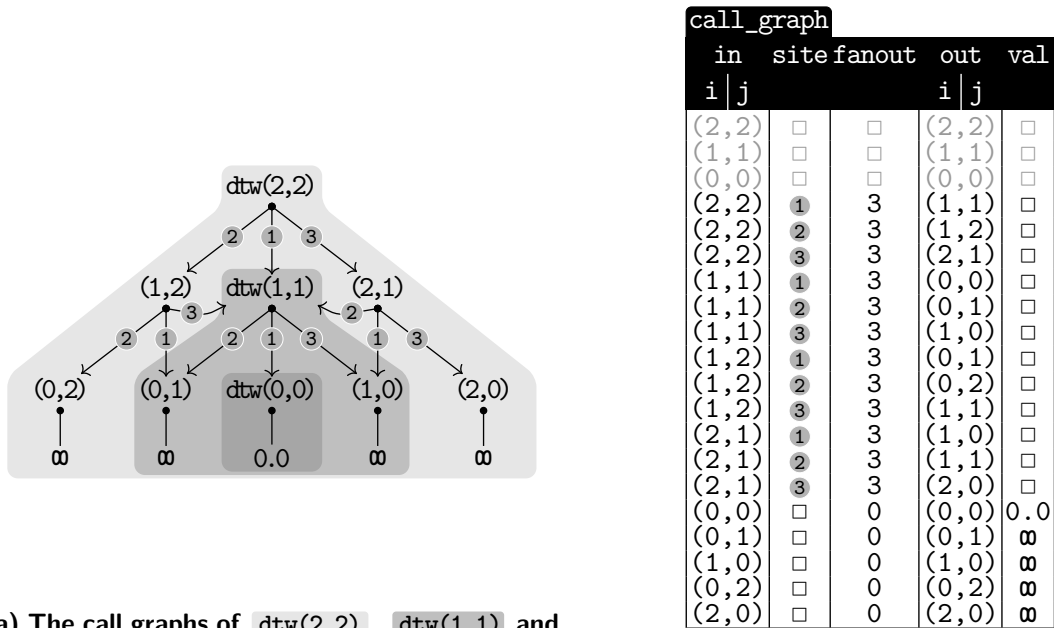
Figure 1.9: Call graphs for two consecutive invocations of UDF `dtw` showcasing memoization.

MEMOIZATION

Compilation allows us to store the results of all invocations of f in a table `memo` (Figure 1.8), which can then be used in future function invocations to trim the call graph size. Figure 1.9 shows the trimmed call graph of `dtw(3,3)` *after* `dtw(2,2)` has been evaluated and its intermediate calls **memoized** earlier. Memoization of recursive calls acts much like base cases and immediately return a value that prunes entire call subgraphs. We discuss memoization in Section 5.5. Automatic memoization based on trimming an explicitly represented call graph contrasts with the programming language implementation of memoization. The latter dynamically discovers a function’s call graph through program or interpreter instrumentation, typically realized in terms of higher-order functions or macro facilities [94, 79]. Since both are unavailable in an RDBMS, we opt for the first-order call graph as data implementation, with no changes to the underlying RDBMS or its query evaluator required.

BATCHING

Developers can expect that a function f is invoked many times independently, say $f(x_1), \dots, f(x_n)$. Enabling **batching** when compiling f changes its signature from $f(x)$ to $f'([x, \dots, x])$. Therefore allowing each individual function invocation $f(x_i)$ to be batched into a single call $f'([x_1, \dots, x_n])$. Lifted function f' constructs a call graph with up to n root nodes and returns a table that maps each argument x_i to its result.



(a) The call graphs of `dtw(2,2)`, `dtw(1,1)` and `dtw(0,0)` collapse into one.

(b) Tabular representation of the call graph in (a).

Figure 1.10: The call graph for invocation of `dtw([(2,2),(1,1),(0,0)])` showcasing batching.

Figure 1.10 shows the call graph of the batched function call `dtw([(2,2),(1,1),(0,0)])`. Note how each call graph of function invocations `(1,1)` and `(0,0)` become subgraphs of the call graph of `(2,2)`. Similar to sharing, the more the constructed call graphs of each argument x_1, \dots, x_n overlap, the smaller the total size of the batched call graph becomes (Section 5.6). Batching is a technique derived from *data-parallel languages* [63]. Another technique, called *flattening*, also bears strong similarities in its concept of lifting functions to arbitrary orders [106]. However, batching is exclusively concerned with first-order flattening: $f : a \rightarrow b$ to $f' : [a] \rightarrow [b]$.

1.4 MORE RELATED WORK

The compilation technique described in this publication is far from the only work done to allow developers to implement complex computations close to the data. Recent research presents other methods for recursive SQL UDF compilation. One such

research effort takes the function body of function f with recursive self-inocations and compiles it, using a form of *continuation-passing style transformation* [65]. This represents an alternative approach for recursive UDF compilation targeting recursive CTEs.

Another method, called *R-SQL* [57], translates SQL UDFs with recursive self-inocation into a set of queries with SELECT and INSERT statements. An external program written in Python repeatedly evaluates these statements until it reaches a fixed point. Consequently, *R-SQL* has to repeatedly cross over the DB/PL line at query runtime, which is precisely what we want to avoid with our present work.

So far, we have focused strictly on SQL UDFs with recursive self-inocations. However, other research efforts exist to identify ways to give developers options to express complex algorithms. Some RDBMS, for example, offer a non-standard *procedural* extension to SQL such as *PL/pgSQL* that ships with PostgreSQL [97]. Its language features were heavily influenced by the PL/SQL language extension of the Oracle database [95]. However, the performance of PL/pgSQL-functions can be disappointing when embedded SQL queries are involved. Such embedded SQL queries require a cost-intensive context switch from extension executor to query executor and back. Compilation rewrites these functions into their semantically equivalent LANGUAGE SQL counterpart targeting recursive CTEs [81, 80].

FunSQL [61] proposes a PL/SQL-like (functional) language which enables developers to write functions in static single assignment form which embeds SQL expressions. Such functions are compiled into a data-flow graph of algebraic operators. We believe SQL itself should be the focus and aim to use it to its full potential, both as the language in which computation is expressed *and* as the compilation target.

Further research was conducted for *HyPer* [92], extending the SQL language itself by adding an ITERATE-operator [96]. This operator allows for an iterative subquery inside a SQL query. The ITERATE-operator works similarly to a *for*-loop, where the developer defines each: initialize, iterate, and stop-expression. In contrast to such research efforts, our approach avoids modifying the underlying RDBM and depends only on standard SQL, which makes it usable without the slow turn-around of patching non-standard changes into the RDBMS.

1.5 RESEARCH FOCUS AND CONTRIBUTION

We aim to measure the viability of functional-style SQL UDFs to implement complex computation. We measure the viability of functional-style UDFs from two angles: *readability* and *runtime performance*. This publication is separated into four parts:

1. The discussion of the user study (Chapter 2) aims to measure the readability and ease of handling functional-style UDFs compared to recursive CTEs.
2. The grammar definition for functional-style UDFs (Chapter 3).
3. The description of the SQL-to-SQL compiler (Chapter 4), which compiles a functional-style UDF using recursive CTEs as their target and removing any recursive self-invocations in the process. Compiling such UDFs improves runtime performance and enables a list of optimization opportunities which we discuss in Chapter 5.
4. The conclusion presents the overall assessment regarding the viability of functional-style UDFs and gives a short outline for future work (Chapter 6).

This work builds upon work previously published by us [70, 71].

2

RECURSION IN SQL - USER STUDY

In 2020, we conducted an anonymous online study tasking developers to solve problems related to functional-style UDFs and recursive CTEs. The study aimed to gauge the readability and ease of handling functional-style UDFs and recursive CTEs comparing each to the other.

The user study targeted three demographics: students that have attended the *Advanced SQL* lecture of summer semester 2020 [56], users subscribed to the *DBWorld mailing list* [68], and interested students, as well as staff of the University of Tübingen, subscribed to the user study mailing list. Students that attended *Advanced SQL* were **not** introduced to functional-style UDFs before this study. The course focused on other SQL features like (recursive) CTEs, window functions, and other features beyond the common `SELECT-FROM-WHERE`-query.

We went through the following steps to conduct the user study: whenever an interested participant contacted us, they received a link with a randomly generated token. They were eligible to participate in the user study exactly once with this token. When the participant submits their result, it is automatically assigned a random identification number, decoupling the submission from the user. Out of 52 interested participants who received a generated token, 19 submitted their results. Dragicevic [69] deems this sufficient to draw meaningful conclusions. The following discussion is

centered around each task's aggregated scores and times (in minutes). Every discussion that centers exclusively around aggregated values is based on a 95% *confidence interval* (CI (95%)) and the p-value (**p-Val**). Skipped tasks do not factor into the statistics.

The study itself is segmented into four general topics: choose the correct implementations (Section 2.1), describe a user-defined function (Section 2.2), manually evaluate a user-defined function (Section 2.3), and implement the *0-1 knapsack* algorithm (Section 2.4). The complete online form of the user study can be found in Appendix A.

User study introduction. Before the user study proper, an introductory text asked each participant *not* to use external programs to run queries found in this study. Instead, we asked them to use pen and paper only, if at all. Then, they had to list three regular programming languages they were familiar with and if they had some exposure to functional programming before this study. This helped us gain insight into the participant's background as a programmer.

We found that the 19 participants form a homogenous group where all are familiar with at least one imperative programming language, the majority being Java and Python, 15 of which had at least some exposure to Functional Programming. Recursive self-invocation and function definition by case distinction are staples of functional programming as it is used in, for example, Haskell [83]. Thus, we assume that most participants are familiar with these concepts, which functional-style UDFs depend on as well.

2.1 CHOOSE THE CORRECT IMPLEMENTATIONS

The first topic gauges the accuracy and speed required of the participants to distinguish between correct and incorrect implementations. The topic is separated into two tasks, each with its own textbook-style algorithm: *Fibonacci Numbers* [78]:

$$\begin{aligned} \text{fib}(0) &= 0 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \end{aligned} \quad (\text{fib})$$

	Task	Points				CI (95%)	p-Val
		min	avg	max	Σ		
fib	Functional-Style UDF	0.0	2.6	4.0	50	[1.89, 3.37]	0.3347
	Recursive CTE	-2.0	2.9	4.0	54	[1.98, 3.80]	
gcd	Functional-Style UDF	0.0	2.5	4.0	48	[1.94, 3.11]	0.2627
	Recursive CTE	-2.0	2.0	4.0	38	[0.92, 3.08]	

(a) Aggregated Scores.

	Task	Time [min]			CI (95%)	p-Val
		min	avg	max		
fib	Functional-Style UDF	1:00	2:47	6:00	[2:03, 3:32]	0.0006
	Recursive CTE	3:00	5:47	12:00	[4:23, 7:11]	
gcd	Functional-Style UDF	1:00	2:16	7:00	[1:37, 2:55]	0.0002
	Recursive CTE	2:00	4:15	8:00	[3:32, 4:58]	

(b) Aggregated Times.

Table 2.1: The aggregated scores and times of the tasks *Fibonacci Numbers* (fib) and *Greatest Common Divisor* (gcd).

Scoring scheme of (a): participants gain 1 point, if a selection is correct and lose 1 point, if incorrect.

and *Greatest Common Divisor* [84]:

$$\begin{aligned} \text{gcd}(n,0) &= n \\ \text{gcd}(n,k) &= \text{gcd}(k,n \% k) \quad . \end{aligned} \tag{gcd}$$

These algorithms were chosen for their concise textbook-style formulation and comparatively simple implementations as recursive CTE and functional-style UDF.

At first, each task presents the participants with the definition of the textbook-style function. Then, they are given a choice of four similar-looking recursive CTEs, and four similar-looking functional-style UDFs for `fib` (and vice versa, for `gcd`). Finally, they are asked to select *only* those snippets that correctly implement the algorithm. The participants also submit the time they need to complete this task.

Scores and times of each task `fib` and `gcd` are aggregated in Tables 2.1a and 2.1b. For each task, the aggregated times for functional-style UDFs are significantly lower than recursive CTEs. One participant skipped both recursive CTE parts of `fib` and `gcd`.

Conclusion. The aggregated scores Table 2.1a suggest that participants perform similarly well when differentiating between correct and incorrect implementations of functional-style UDFs and recursive CTEs. However, participants require only about half the time for functional-style UDFs.

Recursive CTEs disfigure the simple formulation of the textbook-style form, and thus it becomes much more time-consuming to detect subtle errors. Functional-style UDFs, on the other hand, can be compared almost verbatim to the textbook-style form, which is ideal when quickly validating code.

2.2 DESCRIBE A USER-DEFINED FUNCTION

The second topic is separated into two tasks (*Comprehension I* and *Comprehension II*). It gauges the ability and speed with which each participant understands and correctly describes an unknown SQL user-defined function. Both UDFs were chosen for their concise formulation befitting their respective style:

- functional-style UDF `f(i,j,k)` which applies an error that propagates over time (see Figure 2.1), and

```

1 CREATE FUNCTION f(i int, j int, k float)
2 RETURNS float AS $$
3 SELECT CASE
4   WHEN i > j THEN k
5   ELSE (SELECT f(i+1, j, s.b + 0.5 * k)
6         FROM   s
7         WHERE  s.a = i)
8 END;
9 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 2.1: Functional-style UDF $f(i,j,k)$.

```

1 CREATE FUNCTION g(a int)
2 RETURNS bigint AS $$
3 WITH RECURSIVE
4   r(x, y, z) AS (
5     SELECT t.x, t.y, t.z
6     FROM   t
7     WHERE  t.x = a
8     UNION
9     SELECT r.x, t.y, t.z
10    FROM   r, t
11    WHERE  r.y = t.x
12   )
13 SELECT SUM(r.z)
14 FROM   r;
15 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 2.2: Recursive CTE $g(a)$.

```

1 WITH RECURSIVE
2 T( $c_1, \dots, c_n$ ) AS (
3    $q_0$ 
4   UNION
5    $q_c(T)$ 
6 )
7 TABLE T;

```

(a) Recursive CTE.

```

1 u ← DISTINCT( $q_0$ )
2 w ← u
3 LOOP
4   i ← DISTINCT( $q_c(w) \setminus u$ )
5   IF i =  $\emptyset$  THEN BREAK
6   u ← u  $\cup$  i
7   w ← i
8 END
9 RETURN u

```

(b) Pseudo code.

Figure 2.3: The general form of a recursive query T (a) and its semantics described in pseudo code (b). Evaluation requires keeping track of three bag variables: i (*intermediate table*), u (*union table*), and w (*working table*). Figure appeared before in Chapter 1.

- recursive CTE $g(a)$ which traverses a directed graph and sums up each edge weight it passes once (see Figure 2.2).

The participants also submit the time they need to complete this task. Scores and times are aggregated in Tables 2.2a and 2.2b. On average, participants performed better describing functional-style UDF f in about the same time. Two participants skipped the recursive CTE in *Comprehension II*.

Conclusion. The aggregated scores in Table 2.2a show that twice as many participants described the functional-style UDF correctly (i.e. scored two or more points). Over half of the participants had difficulty explaining the recursive CTE and did not discover

Task	Points					Skip	
	0	1	2	3	Σ avg		
Functional-Style UDF	3	4	5	7	35	1.84	0
Recursive CTE	6	5	1	5	22	1.29	2

(a) Aggregated Scores.

Task	Time [min]			CI (95%)	p-Val
	min	avg	max		
Functional-Style UDF	2:00	6:54	12:00	[5:44, 8:03]	0.0497
Recursive CTE	1:00	5:21	11:00	[4:04, 6:38]	

(b) Aggregated Times.

Table 2.2: The aggregated scores and times of tasks *Comprehension I* and *Comprehension II*.

Scoring scheme of (a): Incorrect description (0 points), partially correct description (1 point), correct description without discovering its intended purpose (2 points), correct description of its intended purpose (3 points).

its intended purpose. We point to the convoluted nature of evaluating recursive CTEs to explain why (see Figure 2.3). Thus, it is hard to imagine the participants easily keeping track of all the moving parts of function g and its recursive CTE. Compare this to function f , which has all its functionality explicitly visible. We conclude that developers are more likely to provide a correct description of an unknown functional-style UDF than they would an unknown recursive CTE in roughly the same amount of time.

2.3 MANUALLY EVALUATE A USER-DEFINED FUNCTION

The third topic, *Evaluation*, builds upon the functions f and g (recall Figures 2.1 and 2.2) of the previous tasks *Comprehension I* and *Comprehension II* in Section 2.2. We gauge the ability and speed of the participants to manually evaluate (i.e. without computational aid) each function invocation $f(1,3,0)$ and $g(4)$. The participants then submit the results of these function calls and the time they need to complete this task.

Scores and times of this task are aggregated in Tables 2.3a and 2.3b. On average,

Task	Points					Skip
	0	1	2	Σ	avg	
Functional-Style UDF	3	4	12	28	1.47	0
Recursive CTE	8	0	9	18	1.06	2

(a) Aggregated Scores.

Task	Time [min]			CI (95%)	p-Val
	min	avg	max		
Functional-Style UDF	1:00	3:21	10:00	[2:05, 4:36]	0.2806
Recursive CTE	0:30	2:51	10:00	[1:52, 3:50]	

(b) Aggregated Times.

Table 2.3: The aggregated scores and times of task *Evaluation*.

Scoring scheme of (a): Incorrect result (0 points), partially correct intermediate steps (1 point), the correct result (2 points).

1.	Inititalize <small>Line 1,2</small>			$u = \{r_4\}$	$w = \{r_4\}$
2.	Iterate <small>Line 4,6,7</small>		$i = \{r_3\}$	$u = \{r_4, r_3\}$	$w = \{r_3\}$
3.	Iterate <small>Line 4,6,7</small>		$i = \{r_2\}$	$u = \{r_4, r_3, r_2\}$	$w = \{r_2\}$
4.	Stop <small>Line 4,5</small>		$i = \{ \}$	\hookrightarrow RETURN u	

(a) Bag Variable Trace.

			t
		x	y
r1:	1	2	5
r2:	2	4	3
r3:	3	2	2
r4:	4	3	1

(b) Sample table.

Figure 2.4: The complete variable trace of function g (Figure 2.2) for call $g(4)$. Evaluating a recursive CTE manually requires the developer to keep track of three bag variables i , u , and w (recall Figure 2.3). $\{ \}$ denotes bags of rows.

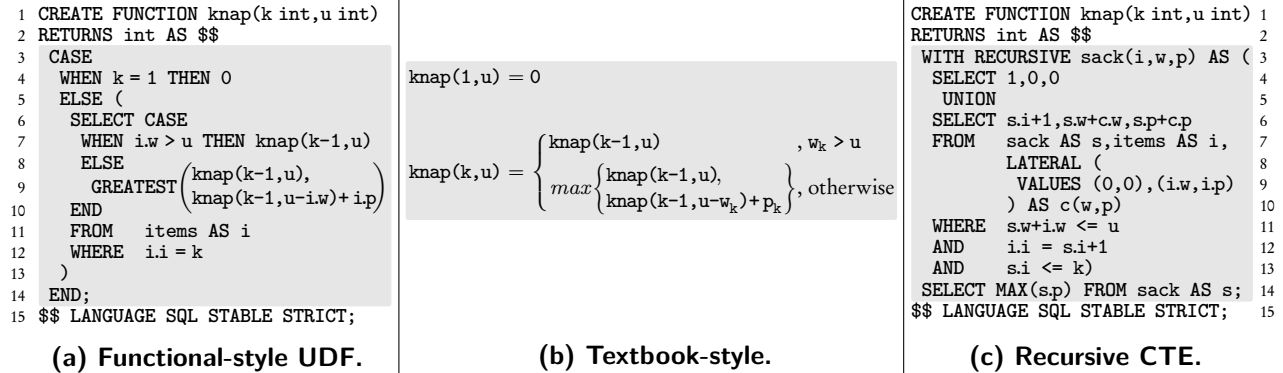


Figure 2.5: Each formulation in (a) – (c) describes the same algorithm to solve the 0-1 Knapsack problem. A rough comparison of the function body in (a) with the body found in (b) shows strong similarities when set side by side. However, compared to (c), the function bodies look almost unrecognizably different. Figure appeared before in Chapter 1.

participants performed better evaluating functional-style UDF f in about the same time. Two participants skipped evaluating the function of *Comprehension II*.

Conclusion. We find that the score of participants describing functions f and g in the previous task (Section 2.2) correlated to the score in this task. Thus, participants that understood a function were almost certainly going to evaluate them correctly.

Furthermore, the way most developers with at least some knowledge about functional programming would evaluate $f(1,3,0)$ is very straightforward: they first keep track of each function call until they reach a base case. Then, they backtrack until they reach the original function call, which returns the final result. This stands in stark contrast with recursive CTEs. Developers have to juggle many implicit variables throughout the evaluation process (recall Figure 2.3). Its complete bag variable trace in Figure 2.4 highlights the cognitive load for a developer to manually evaluate the function call $g(4)$. These results only reinforce the argument that recursive CTEs still put a heavy mental burden on the developers even when a concise formulation is found.

Task	Ratings				Skip
	wrong	minor	syntax	correct	
Functional-Style UDF	3	2	4	2	8
Recursive CTE	-	-	-	-	

(a) Scores.

Task	Time [min]		
	min	avg	max
Functional-Style UDF	3:00	9:06	19:00
Recursive CTE	-	-	-

(b) Aggregated Times.

Table 2.4: The scores and aggregated times of task *0-1 Knapsack*.

Rating scheme of (a): unrecognizable or unfixable (wrong), easily fixable mistakes, e.g., \geq instead of $>$ or missing edge cases (minor), valid but with syntax errors, a compiler would detect, e.g., write `MAX` instead of `GREATEST` or missing `'` (syntax), correct implementation (correct).

2.4 IMPLEMENT THE 0-1 KNAPSACK ALGORITHM

This final topic, *0-1 Knapsack*, asks the participants to choose between writing a functional-style UDF or a recursive CTE. First, the topic presents the participants with the textbook-style definition of the 0-1 Knapsack algorithm [88] (recall Figure 2.5b). The participants then submit their implementation and the time they need to complete this task.

Scores and times of task 0-1 Knapsack are aggregated in Tables 2.4a and 2.4b. No participants even tried implementing the 0-1 Knapsack algorithm as a recursive CTE. Thus, we only discuss aggregated scores and times of the functional-style UDF implementations. Eight participants skipped this task.

Conclusion. We chose the textbook-style 0-1 Knapsack algorithm for this task. Both implementations as functional-style UDF (Figure 2.5a) and recursive CTE (Figure 2.5c) are similar in size, thus making them comparable to each other. Indeed, only the functional-style UDF resembles the textbook-style algorithm.

Combined with the submitted results, we argue that algorithms like 0-1 Knapsack do not lend themselves to be easily rewritten into their equivalent recursive CTE formu-

lation. Thus we conclude that if developers have the choice, they unanimously prefer functional-style UDFs over recursive CTEs when implementing such algorithms.

2.5 SUMMARY

This study aimed to measure the performance of experienced developers working with functional-style UDFs and recursive CTEs. We found that developers require less time distinguishing correct from incorrect functional-style UDFs. They are also more likely to describe and manually evaluate a functional-style UDF accurately. Furthermore, given a choice, they overwhelmingly prefer functional-style UDFs over recursive CTEs to implement the 0-1 Knapsack algorithm. Thus, showing that there exist algorithms, such as these recursive textbook-style algorithms, that developers would prefer in functional-style. Indeed, some algorithms fit recursive CTEs better. Take, for example, the function g (recall Section 2.2), which describes an algorithm applied to a weighted graph. However, we can confidently say that developers appreciate functional-style UDFs as another pillar of support for moving computation close to the data.

3

GRAMMAR OF FUNCTIONAL-STYLE UDFs

We pursue a SQL source-to-source translation that accepts an input UDF f that adheres to the SQL dialect described by the grammar of Figure 3.1. 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`) [97, §38.7]), and
- return values of some scalar type τ or tabular type `TABLE(id, ..., id)` with unique column identifiers id .

Furthermore, function f must be `STRICT` and assumes a *call-by-value* evaluation of f where only actual values are passed as arguments [105]. Indeed, this excludes passing `NULL` as parameters to f , which represents a value currently unknown [66] and thus is incompatible with this restriction. Optional tags `MEMO`, `BATCH` and `ITERATE` allow the user to enable optimizations we discuss in Section 5.5, Section 5.6 and Section 5.4.3 respectively.

The UDF's body is formed top-level by

- set operations that link together two or more subqueries q ,
- tabular conditionals where each branch may be a scalar or tabular subexpression q , and
- a `SELECT-FROM-WHERE` block in which scalar and tabular subexpressions (cf. non-


```

udf ::= CREATE FUNCTION  $f(id, \dots, id)$  RETURNS  $T$  AS
      $$  $q$  $$
      LANGUAGE SQL STABLE STRICT [MEMO] [BATCH] [ITERATE];

 $q$  ::=  $\ell$   $setop(q, \dots, q)$  (n-ary set operator)
      | CASE  $\circledast$ WHEN  $sql$  THEN  $q$  ELSE  $q$  END (tabular conditional)
      | [  $\ell$  WITH  $id$  AS ( $sql$ ), ...,  $id$  AS ( $sql$ ) ] (optional WITH)
      |  $\ell$  SELECT  $e$  AS  $id$ , ...,  $e$  AS  $id$ 
      | [  $\ell$  FROM  $j$  ] (optional FROM)
      | [  $\ell$  WHERE  $e$  ] (optional WHERE)
      | [  $\ell$  GROUP BY  $sql$ , ...,  $sql$  [HAVING  $sql$ ] ] (optional GROUP BY)
      | [  $\ell$  ORDER BY  $sql$ , ...,  $sql$  ] (optional ORDER BY)
      | [  $\ell$  LIMIT  $sql$  ] (optional LIMIT)
      | [  $\ell$  OFFSET  $sql$  ] (optional OFFSET)

 $e$  ::=  $\ell$   $f(sql, \dots, sql)$  (scalar recursive call site)
      | (CASE  $\circledast$ WHEN  $sql$  THEN  $e$  ELSE  $e$  END) (scalar conditional)
      |  $\ell \otimes (e, \dots, e)$  (n-ary operator  $\otimes$ )
      |  $\ell$   $agg(e$  [ORDER BY  $sql$ ]) [OVER ( $over$ )] (aggregate function)
      |  $\ell (q)$  (scalar subquery)
      |  $\ell sql$  (any scalar expression)

 $j$  ::=  $t$  AS  $id$ 
      | ( $j$  [LEFT|FULL] JOIN  $t$  AS  $id$   $\ell$  ON ( $sql$ )) AS  $id$  (JOIN clause)

 $t$  ::=  $\ell$   $f(sql, \dots, sql)$  (tabular recursive call site)
      |  $\ell id$  (table name)
      | [LATERAL]  $\ell (q)$  (tabular subquery)
      |  $\ell sql$  (any tabular expression)

 $setop$  ::= UNION [ALL] (set union)
          | INTERSECT [ALL] (set intersection)
          | EXCEPT [ALL] (set difference)

 $T$  ::=  $\tau$  (scalar SQL type)
      | TABLE( $id, \dots, id$ ) (tabular SQL type)

 $sql$  ::= any SQL expression without recursive call sites
 $over$  ::= OVER clause expression without recursive call sites
 $f$  ::= name of recursive SQL UDF to be compiled
 $agg$  ::= name of aggregate function
 $id$  ::= SQL identifier (table, column, alias, parameter)
 $\ell, \circledast$  ::= unique expression labels

```

Figure 3.1: The grammar for functional-style SQL UDFs. Expression labels (ℓ , \circledast) are internal annotations.

terminals e and j) may nest to arbitrary depth. A variety of optional clauses such as `WITH`, `GROUP BY`, `ORDER BY`, `LIMIT` and `OFFSET` are supported.

The grammar distinguishes scalar (and tabular) expressions that may contain self-invocations (e , q , j , and t) and those that may not (sql and $over$). The non-terminals sql and $over$ function as catch-alls for scalar and tabular subexpressions containing no recursive call sites. The grammar already rules out some queries in which calls to f depend on each other. Ultimately, slicing (Section 4.6) will identify all queries that exhibit such problematic interdependencies. Furthermore, some SQL constructs have been omitted for the sake of simplicity. This does not, however, deter from the expressiveness of functional-style UDFs. Unique labels ℓ and \mathbb{L} are assigned to subexpressions, e.g., a recursive call site. These labels are only internally represented in the parse tree.

4

COMPILING FUNCTIONAL-STYLE UDFs

Let f be a SQL UDF in functional-style that adheres to the grammar defined in Chapter 3. Recursion is expressed in terms of self-inocations of f at, in general, several call sites (cf. ❶ to ❸ in the body of `dtw` in Figure 4.1). The compilation of f replaces its body with SQL code that will evaluate a call, say $f(args)$, in two steps:

1. **Construct call graph** g that originates in root $args$ and records the arguments of all recursive calls that f would perform. Since we do *not* evaluate these calls yet, f 's recursive calls may only depend on $args$ and any other database-wide accessible data, but not on f 's return values. The leaves of g are the non-recursive base cases entered by f (cf. ❹ and ❺ in the body of `dtw` in Figure 4.1).
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 with pseudo code in Sections 4.1 and 4.2 and discuss its SQL representation, which requires the compiler to utilize the well-known *program slicing technique* [108, 104], in Sections 4.3 to 4.6.

The main objective of this chapter is to bridge the gap between the theoretical description and the SQL implementation the compiler emits. For brevity, we keep the SQL implementation described in this chapter quite vanilla, exhibiting restrictions and missing improvements in favor of conciseness. For example, the vanilla

```

1 CREATE FUNCTION dtw(i int, j int)
2 RETURNS real AS $$
3 CASE
4   WHEN i=0 AND j=0 THEN ④0.0
5   WHEN i=0 OR j=0 THEN ⑤∞ -- 'Infinity'::real
6   ELSE (SELECT abs(Z.x - Z.y)
7         +
8         LEAST(①dtw(i-1, j-1),
9              ②dtw(i-1, j ),
10             ③dtw(i , j-1))
11         FROM (X JOIN Y
12              ON ((X.t,Y.t) = (i,j))) AS Z)
13 END;
14 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 4.1: The Dynamic Time Warp (DTW, recall Equation (dtw) in Chapter 1) 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.

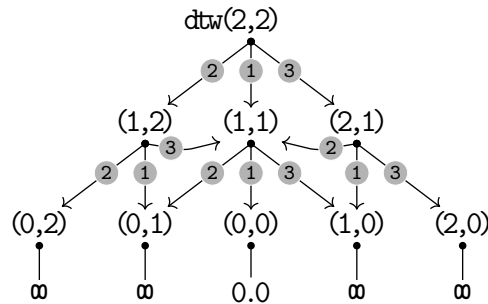


Figure 4.2: The call graph for invocation of $\text{dtw}(2,2)$ (recall Figure 4.1). The call graph describes edges representing either a recursive call *in-site*→*out* or a non-recursive base case *in-val*.

compilation does not yet support functions that exhibit recursive call site invocations dependent on surrounding row variables and table-valued return values. Indeed, in subsequent Chapter 5, we build upon this vanilla SQL implementation to lift these restrictions (see Sections 5.1 and 5.2) and introduce other tweaks and improvements such as linear- and tail-recursion optimization, memoization, and batching.

4.1 STEP 1: CALL GRAPH CONSTRUCTION

The *call graph* provides us with an explicit runtime representation of the work needed to be performed to evaluate $f(args)$. Figure 4.2, for example, shows the graph we

```

call_graph( $f, in, graph$ ):
1   $calls \leftarrow []$  Slices+Calls
2  FOR EACH call site  $site$  of  $f$  that would recursively
   | invoke  $f(out)$  if the arguments are  $in$  DO
3  |  $calls[site] \leftarrow out$ 
4   $edges \leftarrow \emptyset$  Construct
5  IF  $calls \neq []$  THEN
6  | FOR EACH  $(site, out)$  IN  $calls$  DO
7  | | ADD  $in \bullet \text{site} \rightarrow out$  TO  $edges$ 
8  ELSE
9  |  $val \leftarrow$  evaluate body of  $f$  for arguments  $in$ 
10 | ADD  $in \bullet val$  TO  $graph$ 
11  $edges \leftarrow edges \setminus graph$  Invoke
12 FOR EACH  $\bullet \bullet \rightarrow out$  IN  $edges$  DO
13 | ADD  $call\_graph(f, out, graph \cup edges)$  TO  $graph$ 
14 RETURN  $graph$ 

```

Figure 4.3: Call graph construction (pseudo code). Invoked via $call_graph(f, in, \emptyset)$, returns a set of edges.

construct for $dtw(2,2)$. Call graphs contain either of two edges:

- Edges $in \bullet \text{site} \rightarrow out$ embody an invocation with arguments in which leads f to call itself at site $site$ with new arguments out (recall the function body of dtw in Figure 4.1 and its recursive call sites ①, ②, and ③).
- Edges $in \bullet val$ indicate that $f(in)$ enters a non-recursive base case that returns result value val (recall the function body of dtw in Figure 4.1 and its base cases ④ and ⑤).

4.1.1 PSEUDO CODE

Call graph construction can be described as a generic recursive process that accepts a function f and arguments in that returns the call graph. The pseudo code routine $call_graph(f, in, graph)$ of Figure 4.3 does exactly that. Parameter $graph$, initially \emptyset , is used to accumulate the set of edges for the call graph of $f(in)$ and is then returned. This and the following routines are formulated in a style that allows their direct transcription into SQL. You will find the SQL code regions in Sections 4.3 to 4.5 to carry corresponding **labels**:

Slices+Calls With the incoming argument in , we collect the arguments out of all immediate outgoing calls of f (if any) in associative array $calls$. For this, we have to utilize a *sliced* version of f 's body which computes exactly the arguments out

without performing recursive calls. We focus on slicing f in Section 4.6.

Construct If we found that $f(in)$ leads to outgoing calls, construct corresponding edges $in \bullet \text{site} \rightarrow out$ with array *calls* and add them to the call graph. Otherwise, argument in led f into a base case, a case without recursive calls, where we then compute f 's return value val and add $in \bullet val$ 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*. Once the call graph is complete, return it.

Note that $call_graph(f, args, \emptyset)$ constructs 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 looping indefinitely.

4.1.2 OPTIMIZATION: CALL SHARING

call graph size for $dtw(i, i)$

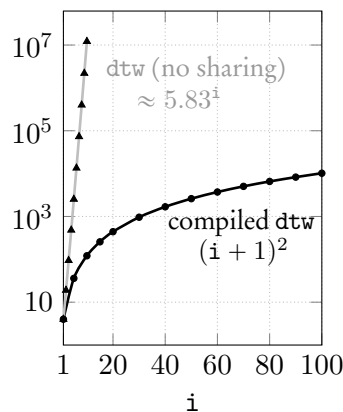


Figure 4.4: Sharing saves function invocations.

Any node in in the call graph — except the root — may have an in-degree greater than one (see node $(1, 1)$ in Figure 4.2, for example). As we assume f to be a pure function without side effects, any call $f(in)$ will yield the same computation. Thus, all evaluation effort of node in and its sub-graphs below may be *shared* by all callers. The $call_graph$ routine implements this sharing by accumulating a *set* of edges. Enabling such sharing can drastically reduce the call graph size and, thus evaluation effort. The naive evaluation of $dtw(i, i)$ with $i > 0$ leads to about $0.87 \times (5.83^i / \sqrt{i})$ recursive calls [77], see \blacktriangle in Figure 4.4. With sharing enabled, the call graph size for $dtw(i, i)$ reduces down to $(i + 1)^2$, see \bullet .

Indeed, we found PostgreSQL does not share the evaluation effort of individual calls (this applies even if uncompiled function f is explicitly marked as being free of side effects [97, §38.7]). Without sharing, the nine inner nodes of the $dtw(2, 2)$ call graph in Figure 4.2 would already unfold into a graph of 19 invocations. In general, PostgreSQL's built-in function evaluation faces dtw call graphs of exponential size (\blacktriangle), which ultimately leads to disastrous function runtimes.

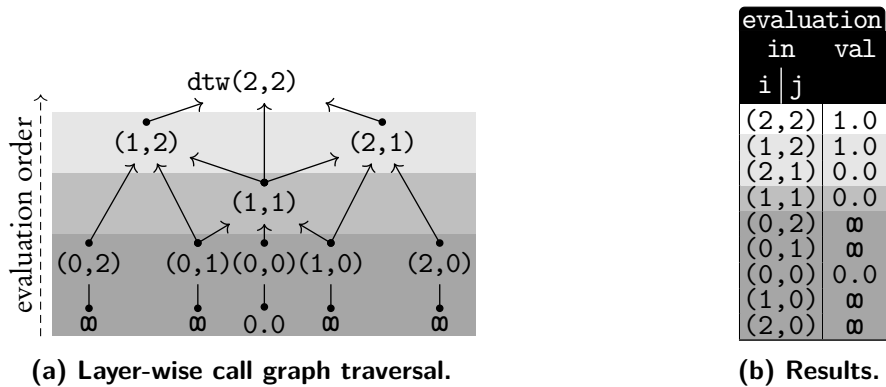


Figure 4.5: A bottom-up call graph traversal populates table `evaluation` with the results of all recursive calls performed during the evaluation of `dtw(2,2)`.

4.2 STEP 2: BOTTOM-UP TRAVERSAL AND EVALUATION

Once the call graph is constructed, the second phase begins evaluation traversing f 's call graph bottom-up. Figure 4.5a depicts this traversal for a sample call graph of `dtw(2,2)` shown in Figure 4.2. The graph is traversed layer-by-layer, starting with the bottommost layer in which the call graph's base case edges $in \leftarrow val$ indicate that $f(in) = val$. We record these discoveries as rows (in, val) in the two-column table `evaluation` (see Figure 4.5b). This table is initially empty but will hold the results of *all* recursive function calls once evaluation is complete. In each evaluation step, a call graph node in with its n recursive calls, $in \leftarrow_{s_1}^{out_1} \dots \leftarrow_{s_n}^{out_n}$, becomes available for evaluation in the next higher layer once all n return values of function calls $f(out_1), \dots, f(out_n)$ are available in table `evaluation`, i.e. if $\{(out_1, val_1), \dots, (out_n, val_n)\} \subseteq \text{evaluation}$. With these return values, we can then evaluate $f(in)$ using a simplified function body where each recursive call site s_i has been replaced by val_i ($i = 1, \dots, n$). This returns value val , which we insert as a new row (in, val) into table `evaluation`. Figure 4.5 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). Once the fourth iteration concludes, table `evaluation` holds row $((2,2), 1.0)$ which ends the evaluation which returns the final result $dtw(2,2) = 1.0$.

```

evaluation( $f, args, e, graph$ ):
1   $go \leftarrow []$  Schedule
2  FOR EACH  $in \bullet \bullet \rightarrow \cdot$  IN  $graph$  SUCH THAT  $e[in] = \square$  DO
3  |  $ret \leftarrow []$ 
4  | FOR EACH SUCH call  $in \bullet \text{site} \rightarrow out$  DO
5  | | IF  $e[out] = \square$  THEN
6  | | | CONTINUE AT 2
7  | |  $ret[site] \leftarrow e[out]$ 
8  |  $go[in] \leftarrow ret$ 
9   $returns \leftarrow []$  Body
10 FOR EACH  $(in, ret)$  IN  $go$  DO
11 |  $val \leftarrow$ 
12 | | evaluate body of  $f$  for arguments  $in$ 
13 | | | with the  $n$ th recursive call site replaced by  $ret[n]$ 
14 |  $returns[in] \leftarrow val$ 
15  $e \leftarrow e \cup returns$  Traverse
16 IF  $e[args] = \square$  THEN
17 | RETURN evaluation( $f, args, e, graph$ )
18 RETURN  $e$ 

```

Figure 4.6: Call graph traversal (pseudo code).

4.2.1 PSEUDO CODE

The pseudo code routine $evaluation(f, args, e, graph)$ of Figure 4.6 realizes the in Section 4.2 described traversal for call graph $graph$ with root node $args$. While we traverse $graph$ bottom-up, we use parameter e to accumulate the table of result rows. Initially, we expect e to only have rows (in, val) that derive from $graph$'s base case edges $in \bullet val$.

Schedule Populate associative array go with those in nodes not yet present in e (i.e. $e[in] = \square$) and have the return values of all outgoing recursive calls ready in e . Store an array ret in $go[in]$ which maps f 's recursive call sites to their return value recorded in e based on their position.

Body For each scheduled node in , evaluate the body of f in which recursive call sites have been replaced with their corresponding return values of array ret 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 $graphs$'s root $args$ is not found in e yet, continue the traversal (on the next higher layer). Otherwise, the evaluation is complete, and e is returned.


```

f(args):
1 graph ← call_graph(f, args, ∅) Graph
2 base_cases ← [] Base
3 FOR EACH in • val IN graph DO
4   | base_cases[in] ← val
5 e ← evaluation(f, args, base_cases, graph) Eval
6 RETURN e[args] Result

```

Figure 4.7: (Pseudo) code to replace the original body of UDF f . A SQL formulation is developed later in this chapter.

4.2.2 COMPILED UDF = CALL GRAPH CONSTRUCTION + BOTTOM-UP TRAVERSAL

To complete compilation of function f , we replace its original body with the composition of call graph construction (Section 4.1) and bottom-up traversal (Section 4.2). Figure 4.7 shows the corresponding pseudo code we will transcribe into proper SQL in Section 4.5. Note how **Base** prepares $base_cases$ using $graph$'s base case edges.

4.3 SQL TEMPLATE: CALL GRAPH CONSTRUCTION

The recursive common table expression of Figure 4.8 computes a tabular encoding of the call graph for $f(\overline{args})$. We use overlines to abbreviate comma-separated lists of columns. Figure 4.9, for example, shows the call graph of $dtw(2,2)$ (Figure 4.9a) and its tabular encoding $call_graph$ (Figure 4.9b). A call edge $in \bullet_{site} \rightarrow out$ is encoded as row $(in, site, fanout, out, \square)$ in which $fanout$ indicates that a call $f(in)$ leads to a total of $fanout$ immediate recursive invocations. For example, $fanout = 3$ characterizes dtw 's 3-fold recursion. Likewise, base case edge $in \bullet_{val}$ maps to row $(in, \square, 0, in, val)$. We use \square to abbreviate SQL's NULL. Furthermore, the cursive *type* in Figure 4.8 indicates template text that needs to be replaced. And, 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 for Figure 4.3:

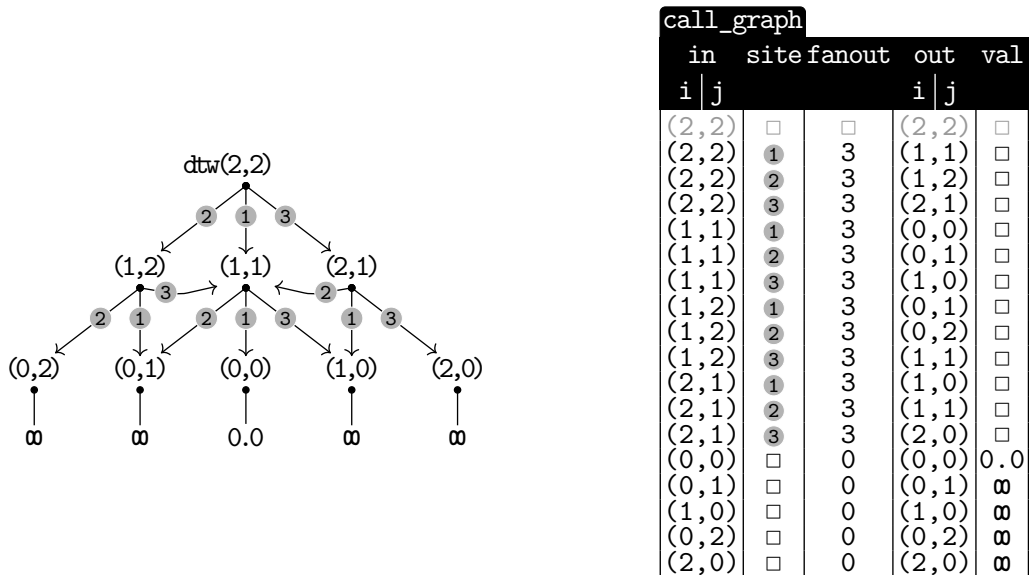
Slices Compute two-column table `slices` in which a row (i, out_i) indicates that the evaluation of $f(\overline{args})$ reaches call site s_i and would invoke $f(out_i)$. If call site s_i is not reached for arguments \overline{args} , record (i, \perp) in `slices` instead. Any distinguishable SQL

```

1 WITH RECURSIVE call_graph(in, site, fanout, out, val) AS (
2   SELECT ROW(f.args), NULL::int, NULL::bigint, ROW(f.args), NULL::τ
3   UNION -- recursive UNION
4   SELECT g.out, edges.*
5   FROM call_graph AS g,
6   LATERAL (
7     WITH slices(site, out) AS (
8       SELECT 1 AS site, out FROM (replace(slice(f, s1), [(g.out).args])) AS _(out) Slices
9       UNION
10      ⋮
11      UNION
12      SELECT n AS site, out FROM (replace(slice(f, sn), [(g.out).args])) AS _(out)
13     ),
14     calls(site, fanout, out, val) AS (
15       SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL::τ AS val
16       FROM slices AS s
17       WHERE s.out <> ⊥ Calls
18     )
19     TABLE calls
20     UNION ALL
21     SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out,
22           (replace(body(f, [NULL::τ, ..., NULL::τ]),
23                   [(g.out).args])) AS val
24     WHERE NOT EXISTS (TABLE calls) Construct
25   ) AS edges(site, fanout, out, val)
26 WHERE g.fanout > 0 OR g.fanout IS NULL
27 ) Invoke

```

Figure 4.8: Call graph construction (SQL template, compare with pseudo code in Figure 4.3).



(a) The call graph describes edges representing either a recursive call $in \leftarrow site \rightarrow out$ or a non-recursive base case $in \leftarrow val$.

(b) Tabular representation of the call graph in (a). We use \square to abbreviate SQL's NULL.

Figure 4.9: The call graph for invocation of $dtw(2,2)$.

```

replace(slice(dtw, ①), [i, j]) =
1 CASE
2   WHEN i=0 AND j=0 THEN ④ ⊥
3   WHEN i=0 OR j=0 THEN ⑤ ⊥
4   ELSE (SELECT ① (SELECT ROW(i-1, j-1))
5         FROM (X JOIN Y
6              ON ((X.t, Y.t) = (i, j))) AS Z)
7 END;

```

Figure 4.10: 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. Then, all references to `dtw`'s arguments are replaced by `i` and `j`.

value may be used to represent \perp ("bottom"). Table slices will carry n rows, if f has n recursive call sites. For `dtw`, this would be $n = 3$, where $s_i = \textcircled{i}$. To obtain out_i , we evaluate $replace(slice(f, s_i), [x_1, x_2, \dots])$. $slice(f, s_i)$ produces a *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 [108, 104, 107] is an established code transformation technique that we adapt for SQL in subsequent Section 4.6. Finally, $replace(f, [x_1, x_2, \dots])$ produces out_i , where all references to f 's arguments \overline{args} in f 's body are replaced with x_1, x_2, \dots . Figure 4.10, for example, shows the result of $replace(slice(dtw, \textcircled{1}), [i, j])$.

Calls For each recursive call site s_i that has been reached, collect (the tail of) its call graph edge $\bullet \xrightarrow{s_i} out_i$ in table `calls`. We use aggregate `COUNT(*) OVER ()` [97, §4.2.8] over the non-empty slices to find the number of recursive calls (fanout) performed by $f(\overline{args})$.

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 of scalar type τ . Construct (the tail of the) base case edge $\bullet \dashrightarrow val$. We use $replace(body(f, [e_1, e_2, \dots]), [x_1, x_2, \dots])$ to modify the body of f . $body(f, [e_1, e_2, \dots])$ replaces the call sites of f with expressions e_1, e_2, \dots and $replace(f, [x_1, x_2, \dots])$ takes all references to the arguments of f in its body and replaces them with x_1, x_2, \dots . Figure 4.11, for example, shows the result of $replace(body(f, [e_1, e_2, e_3]), [i, j])$. Since the recursive call sites are irrelevant in the base case, the template sets e_i to `NULL` of type τ .

Invoke Add edges to the call graph in each iteration. Proceed with the call graph construction using the arguments out of the added graph edges g from the directly

```

replace(body(f, [e1, e2, e3]), [i, j]) =
1 CASE
2   WHEN i=0 AND j=0 THEN ④ 0.0
3   WHEN i=0 OR j=0 THEN ⑤ '∞' -- 'Infinity'::real
4   ELSE (SELECT abs(Z.x - Z.y)
5         +
6         LEAST(① e1,
7              ② e2,
8              ③ e3)
9         FROM (X JOIN Y
10              ON ((X.t, Y.t) = (i, j))) AS Z)
11 END;

```

Figure 4.11: Body of UDF `dtw` with its call sites and arguments replaced by e_1, e_2, e_3 and i, j , respectively.

previous iteration as arguments to f . As per the semantics of the recursive UNION, the construction will terminate once no new graph edges are discovered, i.e. all edges are base cases with `fanout = 0`.

4.4 SQL TEMPLATE: BOTTOM-UP TRAVERSAL

The SQL template of Figure 4.12 realizes the layer-by-layer call graph traversal as introduced in Figure 4.6. 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 found in `call_graph` (recall Figure 4.5):

Schedule Identify unevaluated nodes g whose recursive calls (of which there are $g.fanout$ many) are all found in table `evaluation`. Like the pseudo code, we collect the return values of these recursive calls in an SQL array `ret` using the custom aggregate `array_gather(v, i)` which builds array a where $a[i] = v$. Subsequent tweaks and improvements to the template in Chapter 5 will not require `array_gather(v, i)` anymore.

Note that throughout the bottom-up traversal, each row in the `evaluation` CTE is joined with the `call_graph` CTE, prompting a full CTE scan every time. Thus, the runtime complexity of compiled function f is $|evaluation| \times |call_graph|$, where $|evaluation| > |call_graph|$ i.e. the lion share of the total execution time

```

1 [WITH RECURSIVE] evaluation(in,val) AS (
2 TABLE base_cases
3 UNION ALL -- recursive UNION ALL
4 (WITH e(in,val) AS (TABLE evaluation),
5 returns(in,val) AS (
6 SELECT go.in,
7     (replace(body(f, [go.ret[1], ..., go.ret[n]],
8     [(go.in).args])) AS val
9 FROM (SELECT g.in, array_gather(e.val,g.site) AS ret
10 FROM call_graph AS g, e
11 WHERE g.out = e.in
12 AND NOT EXISTS (SELECT FROM e WHERE e.in = g.in)
13 GROUP BY g.in, g.fanout
14 HAVING COUNT(*) = g.fanout
15 ) AS go(in,ret)
16 )
17 SELECT results.*
18 FROM (TABLE e UNION ALL TABLE returns) AS results(in,val)
19 WHERE NOT EXISTS (SELECT FROM e WHERE e.in = ROW(f.args))
20 )
21 )

```

Figure 4.12: Bottom-up call graph traversal (SQL template, compare with pseudo code in Figure 4.6).

to evaluate function f . Indeed, depending on function f , the size of evaluation can be *many* times the size of `call_graph`. For example, measuring the size of `call_graph` and evaluation for invocation of compiled function `dtw(100,100)` yields $|\text{evaluation}| = 108,721$ and $|\text{call_graph}| = 10,201$. Chapter 5 discusses ways to improve this through reference counting.

Body For each node `go`, evaluate the body of f with its call sites replaced by the return values found in `go.ret`. We use `replace(body(f, [e1, e2, ...]), [x1, x2, ...])` to modify the body of UDF f in which `body(f, [e1, e2, ...])` replaces the call sites of f with expressions e_1, e_2, \dots and `replace(f, [x1, x2, ...])` replaces all references to the arguments of f in its body with x_1, x_2, \dots . Figure 4.11, for example, shows the result of `replace(body(f, [e1, e2, e3]), [i, j])`. Record the found results in table `returns(in, val)`.

Traverse Add returns to evaluation to form the overall known results so far. The CTE will continue to iterate until the result for argument $f.args$ is finally found in results.

```



1 CREATE FUNCTION  $f(\overline{\text{args}})$  RETURNS  $\tau$ 
2 AS $$
3   WITH RECURSIVE call_graph(in,site,fanout,out,val) AS (
4     <see Figure 4.8>
5   ),
6   base_cases(in,val) AS (
7     SELECT g.in, g.val
8     FROM   call_graph AS g
9     WHERE  g.fanout = 0
10  ),
11  evaluation(in,val) AS (
12    <see Figure 4.12>
13  )
14  SELECT e.val
15  FROM   evaluation AS e
16  WHERE  e.in = ROW( $f.\overline{\text{args}}$ );
17 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 4.13: Compiled SQL code replaces the functional-style UDF f . (Compare with pseudo code in Figure 4.7.)

Note that the code regions **Schedule** and **Traverse** in Figure 4.12 hint at a non-monotonic SQL query through its usage of aggregates and NOT EXISTS that some RDBMSs rule out syntactically if they appear in a recursive CTE. However, line 4 of Figure 4.12 references evaluation exactly once and stores only its most recent rows into CTE e, which works around this problem. We find that the templates discussed here are accepted by postgresSQL [97], HyPer [92], or Umbra [93], for example.

4.5 EMITTING CODE

The compiler completes its job by emitting the SQL function f of Figure 4.13 which glues the two SQL templates of Sections 4.3 and 4.4 together. Just like the pseudo code of Figure 4.7, **Graph** and **Base** prepare the call graph and base case tables as expected by **Eval**. From table evaluation, **Result** extracts f 's return value for arguments $\overline{\text{args}}$ to deliver the function's final result. This purely CTE-based form of f serves as the drop-in replacement for the original functional-style UDF and already improves the runtimes compared to the uncompiled function (see  and  in Figure 4.14).

This vanilla approach does not yet enable recursive call sites to reference surrounding row variables, table-valued return types, and other runtime optimizations. In subse-

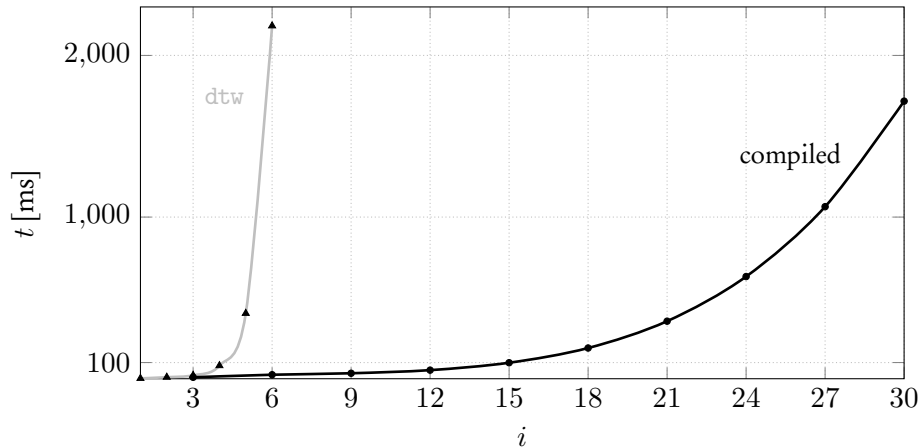


Figure 4.14: Evaluating $\text{dtw}(i, i)$. Compares the runtimes before and after compilation.

quent Chapter 5, we lift these restrictions and further exploit this call graph-centric approach to UDF compilation. Besides the opportunity to share calls and thus evaluation effort, we find that

- some functions lead to simpler call graphs (i.e. call chains), which the compiler can use to optimize code (see Sections 5.3 and 5.4).
- Traversal-based evaluation creates a table filled with the results of all *intermediate recursive calls*. Future calls to f can benefit when these are stored for future use (see Section 5.5).
- Call graph construction allows the user to batch together many function invocations into one enabling sharing beyond the boundaries of a single function call (see Section 5.6).

4.6 SLICING FUNCTIONAL-STYLE SQL UDFs

With $\text{slice}(f, s_i)$, we are after a minimal version of function f 's original body – a *slice* – which still retains those expressions relevant to the evaluation of the argument expressions at call site with label s_i . All argument expressions of s_i must evaluate the same in the sliced body as they would in the original. This closely resembles *program slicing* [108] in which a program is reduced to contain only statements relevant to executing a statement s called *slicing criterion*. We adapt slicing to aid in compiling

functional-style UDFs (recall use of $slice(f, s_i)$ in SQL template in Figure 4.8).

4.6.1 EVALUATION PATHS

In the statement-by-statement execution of an imperative program, we can mechanically identify the *trace* of statements [58] which are required for evaluating 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 subexpression of e_1 (in this case, the evaluation of e_2 is done before e_1), or
- e_1 binds a variable that is in scope of e_2 , or
- e_1 is a predicate that may inhibit the evaluation of e_2 .

Given a query q , we use $q \bullet \rightarrow \{p_1, \dots, p_n\}$ to compute its set of *evaluation paths*. Each path p_i is a sequence of expression labels $[e, \dots]$ (recall grammar in Chapter 3) which stand in for their associated expressions e_ℓ . Label $\mathbf{1}$ precedes $\mathbf{2}$ in p_i (or: $\mathbf{1} <_{p_i} \mathbf{2}$), if e_1 is entered before e_2 in q . Figure 4.15 defines $\bullet \rightarrow$ in terms of inference rules that inspect the syntax of q .

Note that, for brevity, we write the elements of the FROM-list as $T_1 \mathbin{\textcircled{;}} \dots \mathbin{\textcircled{;}} T_f$ which recursively expands to

$$\begin{aligned} T_1 &\equiv t_1 \text{ AS } id_{21} \\ T_1 \mathbin{\textcircled{;}} T_2 &\equiv (T_1 \text{ [LEFT|FULL] JOIN } t_2 \text{ AS } id \text{ } \overset{N_2}{\text{ON}} \text{ } sql) \text{ AS } id_{22} \\ T_1 \mathbin{\textcircled{;}} \dots \mathbin{\textcircled{;}} T_{f-1} \mathbin{\textcircled{;}} T_f &\equiv (T_1 \mathbin{\textcircled{;}} \dots \mathbin{\textcircled{;}} T_{f-1} \text{ [LEFT|FULL] JOIN } t_f \text{ AS } id \text{ } \overset{N_f}{\text{ON}} \text{ } sql) \text{ AS } id_{2f} . \end{aligned}$$

Figure 4.16 superimposes the evaluation paths of UDF dtw on its body query. We see that evaluation path $[\mathbf{A}, \mathbf{4}]$ contains the expressions entered before base case literal 0.0 (label $\mathbf{4}$) is evaluated. One evaluation path leading to call site $\mathbf{1}$ is $p = [\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H}, \mathbf{I}, \mathbf{K}, \mathbf{1}]$. We find the FROM clause (\mathbf{D}) and table X (\mathbf{E}) and table Y (\mathbf{F}) in p as these bind row variables X and Y to ON clause (\mathbf{G}). Its predicate controls whether $\mathbf{1}$ is evaluated for a given variable binding, thus we find $\mathbf{G} <_p \mathbf{1}$ as well.

Note that evaluation paths reflect dependencies between the query subexpression prescribed by the SQL semantics [101] and are independent of any particular order an

$$\begin{array}{c}
\overline{\ell f(sql_1, \dots, sql_n) \bullet \rightarrow \{[\ell]\}} \quad \overline{\ell sql \bullet \rightarrow \{[\ell]\}} \quad \overline{\ell id \bullet \rightarrow \{[\ell]\}} \\
\\
\frac{q \bullet \rightarrow \pi}{\ell(q) \bullet \rightarrow \{[\ell]\} \oplus \pi} \quad \frac{e_i \bullet \rightarrow \pi_i|_{i=1..n}}{\ell \otimes (e_1, \dots, e_n) \bullet \rightarrow \{[\ell]\} \oplus \bigcup_{i=1..n} \pi_i} \\
\\
\frac{e_1 \bullet \rightarrow \pi_1 \quad e_2 \bullet \rightarrow \pi_2}{\text{CASE } \ell \text{ WHEN } sql \text{ THEN } e_1 \\ \text{ELSE } e_2 \bullet \rightarrow \{[\ell]\} \oplus (\pi_1 \cup \pi_2) \\ \text{END}} \quad \frac{q_1 \bullet \rightarrow \pi_1 \quad q_2 \bullet \rightarrow \pi_2}{\text{CASE } \ell \text{ WHEN } sql \text{ THEN } q_1 \\ \text{ELSE } q_2 \bullet \rightarrow \{[\ell]\} \oplus (\pi_1 \cup \pi_2) \\ \text{END}} \\
\\
\frac{q_i \bullet \rightarrow \pi_i|_{i=1..n}}{\ell \text{ setop}(q_1, \dots, q_n) \bullet \rightarrow \{[\ell]\} \oplus \bigcup_{i=1..n} \pi_i} \quad \frac{e \bullet \rightarrow \pi}{\ell \text{ agg}(e \text{ ORDER BY } sql) \\ \text{OVER } (over) \bullet \rightarrow \{[\ell]\} \oplus \pi} \\
\\
\frac{e_i \bullet \rightarrow \pi_{1i}|_{i=1..s} \quad t_j \bullet \rightarrow \pi_{2j}|_{j=1..f} \quad e \bullet \rightarrow \pi}{\begin{array}{l} \mathcal{T} \text{ WITH } id_1 \text{ AS } (sql_1), \dots, id_t \text{ AS } (sql_t) \\ \mathcal{S} \text{ SELECT } e_1 \text{ AS } id_{11}, \dots, e_s \text{ AS } id_{1s} \\ \mathcal{F} \text{ FROM } T_1 \text{ ; } \dots \text{ ; } T_f \\ \mathcal{W} \text{ WHERE } e \\ \mathcal{G} \text{ GROUP BY } sql_1, \dots, sql_g \text{ HAVING } sql \\ \mathcal{O} \text{ ORDER BY } sql_1, \dots, sql_o \\ \mathcal{L} \text{ LIMIT } sql \\ \mathcal{E} \text{ OFFSET } sql \end{array} \bullet \rightarrow \{[\mathcal{T}, \mathcal{E}]\} \oplus \pi_{21} \oplus \pi_{22} \oplus \{[\mathcal{N}_2]\} \dots \oplus \pi_{2j} \oplus \{[\mathcal{N}_j]\} \oplus \\ \{[\mathcal{W}]\} \oplus \pi \oplus \{[\mathcal{G}, \mathcal{O}, \mathcal{L}, \mathcal{E}, \mathcal{S}]\} \oplus \bigcup_{i=1..s} \pi_{1i}}
\end{array}$$

Figure 4.15: $q \bullet \rightarrow \pi$ derives the set π of evaluation paths for SQL query q . Operator \oplus combines two path sets: $\pi_1 \oplus \pi_2 = \{p_1 \parallel p_2 \mid p_1 \in \pi_1, p_2 \in \pi_2\}$ where \parallel denotes path concatenation. Any $T_j|_{j=1..f}$ uniquely identifies tabular expression t_j (and label \mathcal{N}_j , if $j > 1$).

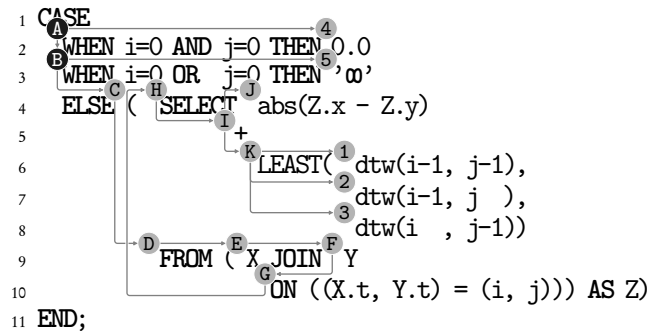


Figure 4.16: (Prefix tree of) Evaluation paths of UDF dtw superimposed on its body. [A, B, C, ...] shown as A-B-C-....

RDBMS may choose to evaluate q . This also means the evaluation path is produced independent of any alterations that specific query optimizations would have applied.

$$\begin{aligned}
\mathbb{L} f(sql_1, \dots, sql_n) \Downarrow_C &= \begin{cases} \mathbb{L} (\text{SELECT ROW}(sql_1, \dots, sql_n)) & \text{if } \mathbb{L} \in C \\ \perp & \text{otherwise} \end{cases} & (\text{REC}) \\
\mathbb{L} sql \Downarrow_C &= \begin{cases} \mathbb{L} sql & \text{if } \mathbb{L} \in C \\ \perp & \text{otherwise} \end{cases} & (\text{SQL}) \\
\mathbb{L} \otimes(e_1, \dots, e_n) \Downarrow_C &= \begin{cases} e_i \Downarrow_C & \text{if } E = \{i\} \\ \perp & \text{otherwise} \end{cases} \quad \text{with } E = \{i \in 1 \dots n \mid e_i \Downarrow_C \neq \perp\} & (\text{OP}) \\
\mathbb{L} (q) \Downarrow_C &= \mathbb{L} (q \Downarrow_C) & (\text{SUB}) \\
\mathbb{L} id \Downarrow_C &= \begin{cases} \mathbb{L} id & \text{if } \mathbb{L} \in C \\ \perp & \text{otherwise} \end{cases} & (\text{TBL}) \\
\text{CASE } \mathbb{L} \text{ WHEN } sql \text{ THEN } e_1 & \Downarrow_C = \begin{cases} (\text{SELECT * FROM (SELECT } e_1 \Downarrow_C \text{) AS _ WHERE } sql \\ \mathbb{L} \text{ UNION ALL} \\ \text{SELECT * FROM (SELECT } e_2 \Downarrow_C \text{) AS _ WHERE NOT } sql \end{cases} & (\text{CASE}_e) \\
\text{ELSE } e_2 & \\
\text{END} & \\
\text{CASE } \mathbb{L} \text{ WHEN } sql \text{ THEN } q_1 & \Downarrow_C = \begin{cases} (\text{SELECT * FROM (} q_1 \Downarrow_C \text{) AS _ WHERE } sql \\ \mathbb{L} \text{ UNION ALL} \\ \text{SELECT * FROM (} q_2 \Downarrow_C \text{) AS _ WHERE NOT } sql \end{cases} & (\text{CASE}_q) \\
\text{ELSE } q_2 & \\
\text{END} & \\
\mathbb{L} setop(q_1, \dots, q_n) \Downarrow_C &= \begin{cases} q_i \Downarrow_C & \text{if } Q = \{i\} \\ \perp & \text{otherwise} \end{cases} \quad \text{with } Q = \{i \in 1 \dots n \mid e_i \Downarrow_C \neq \perp\} & (\text{SET}) \\
\mathbb{L} agg(e \text{ ORDER BY } sql) & \Downarrow_C = (e \Downarrow_C) & (\text{AGG}) \\
\text{OVER (over)} &
\end{aligned}$$

Figure 4.17: $q \Downarrow_C$ ($e \Downarrow_C$) slices SQL query q (expression $e \Downarrow_C$) to expose the arguments of the recursive call at the call site identified by C . Continued in Figure 4.18.

$$\begin{array}{l}
\mathbb{T} \text{ WITH } id_1 \text{ AS } (sql_1), \dots, id_t \text{ AS } (sql_t) \\
\mathbb{S} \text{ SELECT } id_{1i}. * \\
\mathbb{F} \text{ FROM } T_1 \mathbin{\&}; \dots \mathbin{\&} T_f, \text{ LATERAL } (e_i \Downarrow_C) \text{ AS } id_{1i} \\
\mathbb{W} \text{ WHERE } e \\
\mathbb{G} \text{ GROUP BY } sql_1, \dots, sql_g \text{ HAVING } sql \\
\mathbb{O} \text{ ORDER BY } sql_1, \dots, sql_o \\
\mathbb{L} \text{ LIMIT } sql \\
\mathbb{E} \text{ OFFSET } sql \\
\text{if } \{s, f, w\} \subseteq C, \\
E = \{i\} \quad (\text{SELECT})
\end{array}
=
\begin{array}{l}
\mathbb{T} \text{ WITH } id_1 \text{ AS } (sql_1), \dots, id_t \text{ AS } (sql_t) \\
\mathbb{S} \text{ SELECT } e_1 \text{ AS } id_{11}, \dots, e_s \text{ AS } id_{1s} \\
\mathbb{F} \text{ FROM } T_1 \mathbin{\&}; \dots \mathbin{\&} T_f \\
\mathbb{W} \text{ WHERE } e \\
\mathbb{G} \text{ GROUP BY } sql_1, \dots, sql_g \text{ HAVING } sql \\
\mathbb{O} \text{ ORDER BY } sql_1, \dots, sql_o \\
\mathbb{L} \text{ LIMIT } sql \\
\mathbb{E} \text{ OFFSET } sql \\
\text{if } \{f, w\} \subseteq C, \\
E = \{i\} \quad (\text{WHERE})
\end{array}
=
\begin{array}{l}
\mathbb{T} \text{ WITH } id_1 \text{ AS } (sql_1), \dots, id_t \text{ AS } (sql_t) \\
\mathbb{S} \text{ SELECT } id_{2j}. * \\
\mathbb{F} \text{ FROM } T_1 \mathbin{\&}; \dots \mathbin{\&} T_{j-1}, t_j \Downarrow_C \text{ AS } id_{2j} \\
\text{if } f \in C, \\
T = \{j\} \quad (\text{FROM})
\end{array}
=
\begin{array}{l}
\mathbb{T} \text{ WITH } id_1 \text{ AS } (sql_1), \dots, id_t \text{ AS } (sql_t) \\
\mathbb{S} \text{ SELECT } \perp \\
\mathbb{F} \text{ FROM } T_1 \mathbin{\&}; \dots \mathbin{\&} T_f \\
\mathbb{W} \text{ WHERE } e \\
\mathbb{G} \text{ GROUP BY } sql_1, \dots, sql_g \text{ HAVING } sql \\
\mathbb{O} \text{ ORDER BY } sql_1, \dots, sql_o \\
\mathbb{L} \text{ LIMIT } sql \\
\mathbb{E} \text{ OFFSET } sql \\
\text{otherwise} \\
(\text{SFW})
\end{array}$$

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

Figure 4.18: $q \Downarrow_C$ also slices SELECT-FROM-WHERE to expose the arguments of the recursive call at the call site identified by C . Any $T_j|_{j=1..f}$ uniquely identifies tabular expression t_j (and label \mathbb{N} , if $j > 1$). Continued from Figure 4.17.

4.6.2 CALL SITE SLICES

We build upon evaluation paths (from previous Section 4.6.1) and define $slice(f, s_i)$ as follows:

- (1) Let q denote the query body of UDF f . Derive its set π of evaluation paths via $q \bullet \rightarrow \pi$. Then, let $\pi[s_i] \subseteq \pi$ hold the subset of paths that contain call site label s_i .
- (2) The labels in $C = \{s_i\} \cup \bigcup_{p \in \pi[s_i]} \{c \mid c <_p s_i\}$ are those expressions in q that are entered before s_i on some evaluation path. To ensure that the resulting slice does not depend on other self-inocations of f , we impose the syntactic restriction that C may not contain call site labels other than s_i .
- (3) Find and return the sliced query $q^- = q \Downarrow_C$, which is stripped of expressions irrelevant to evaluating the arguments at call site s_i . $\cdot \Downarrow_C$ is defined in Figures 4.17 and 4.18 and discussed in further detail below.

SQL transformation $q \Downarrow_C$ is defined by syntactic case analysis on q guided by set C . The slicing replaces any irrelevant subexpression with \perp , an arbitrary yet unique SQL value which signifies that the evaluation of q has *not* entered call site s_i .

REC If this is the call site we are after (i.e. $\bullet \in C$), remove the recursive call to f and only keep its n arguments packaged inside a row constructor inside a SELECT-clause. Otherwise, the expression is irrelevant.

SQL Do not descend further into any SQL expression sql which does not contain a call site (recall Figure 3.1).

OP If the call site found in the i th argument of n -ary operator \otimes (meaning $e_i \Downarrow_C \neq \perp$ and thus $E = \{i\}$), keep slicing that argument e_i only. Otherwise, discard the operator entirely.

CASE _{e/q} Preserve the branches e_1 and e_2 (or q_1 and q_2) such that they are (not) evaluated under the same conditions as the original query. In any case, we transform the CASE-clause into its equivalent SELECT-WHERE-clauses as part of this process, lifting the scalar conditional expression into its tabular representation. Recursive application of $\cdot \Downarrow_C$ discards irrelevant branches with \perp .

SET If the call site found in the i th argument of n -ary set-operator $setop$ (meaning $q_i \Downarrow_C \neq \perp$ and thus $Q = \{i\}$), keep slicing that argument q_i . Otherwise, discard the operator entirely.

SELECT For a call site found in the SELECT expression e_i , discard any other expression.

Then, move e_i into a LATERAL subquery appended with a cross-join ($,$) at the end of FROM $T_1 \text{ ; } \dots \text{ ; } T_f$. The LATERAL subquery is assigned a unique row variable id . Expression $id.*$ replaces all expressions in the SELECT-clause to extract and observe the call's argument values. This lifts e_i from a (possibly) scalar expression into a tabular context.

WHERE Similar to SELECT, except we append the WHERE predicate e at the end of FROM $T_1 \text{ ; } \dots \text{ ; } T_f$.

FROM For a call site found in the FROM tabular expression t_j ($j \in 1, \dots, f$), we keep only $T_1 \text{ ; } \dots \text{ ; } T_{j-1}$ and append t_j with a cross-join ($,$) at the end of it. Then, we move $id_{2j}.*$ into the SELECT-clause where we extract and observe the call's argument values.

```

slice(dtw, ①) =
1  CASE
2  ① WHEN i=0 AND j=0 THEN ④ ⊥
3  ② WHEN i=0 OR j=0 THEN ⑤ ⊥
4  ELSE ③ ( ⑥ SELECT ① SELECT ROW(i-1, j-1)
5          ⑦ FROM ( ⑧ X JOIN ⑨ Y
6                ⑩ ON ((X.t, Y.t) = (i, j))) AS Z)
7  END;

```

Figure 4.19: Slice of the body of UDF `dtw` for call site ①. Subexpressions irrelevant to the computation for the arguments $i-1, j-1$ at call site ① have been removed.

For example, the Figure 4.19 shows the result of $slice(dtw, ①)$. For readability sake, we refrain from rewriting the CASE as its SELECT-WHERE-clause (recall $CASE_e$ in Figure 4.17). The set C is $\{①, ②, ③, ④, ⑤, ⑥, ⑦, ⑧, ⑨, ⑩, ①\}$. The result is a valid SQL slice that either returns the arguments $ROW(i-1, j-1)$ of the call at site with label ① or \perp if the invocation of `dtw` enters a base case branch (④ or ⑤).

Now that we described the vanilla compilation method and its implementation, we can tweak and improve it in Chapter 5. There, we will lift restrictions and exploit the call graph for further optimizations.

5

TWEAKS AND IMPROVEMENTS

The vanilla compilation method described in previous Chapter 4 already has a positive impact on the execution times of functional-style UDFs. This chapter introduces tweaks and improvements to this vanilla compilation technique to lower execution times even further as well as lift syntactical restrictions previously imposed. Indeed, as Figure 5.1 indicates, some of these improvements to the compiled functional-style UDF `dtw` (🔪) moves execution times close to the runtime performance of a hand-crafted and carefully optimized CTE (🔪).

In Section 5.1, we ultimately replace the vanilla evaluation CTE with one that applies reference counting to the bottom-up traversal process and lifts the restriction where recursive call sites were not allowed to reference surrounding row variables. Further improvements in all subsequent sections build upon this tweaked SQL template. In Section 5.2, we describe necessary tweaks to the SQL template that enables compilation of functional-style UDFs with table-valued return type. In Sections 5.3 and 5.4, if the compiler identifies the functional-style UDF as linear- or tail-recursive, we can further simplify the SQL templates the compiler produces. Finally, we introduce two more optimizations developers can enable when compiling functional-style UDFs:

- **Memoization** retains the results of all *intermediate recursive calls* for future calls (Section 5.5), and

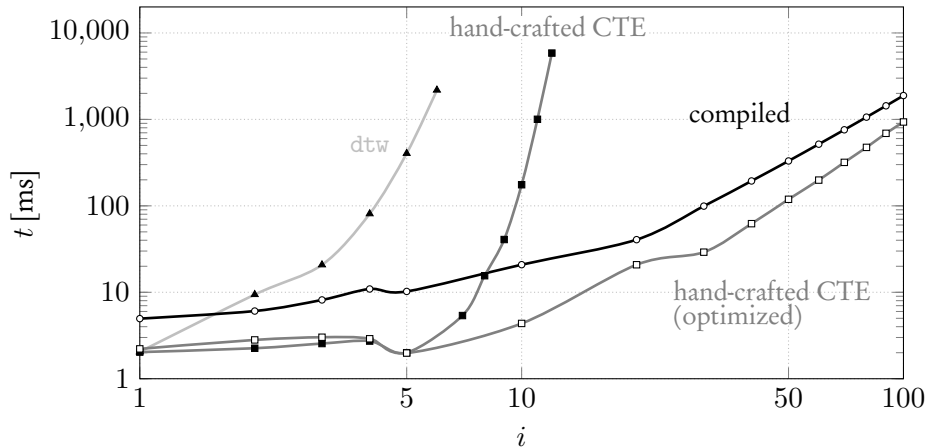


Figure 5.1: Evaluating $\text{dtw}(i, i)$: Execution time comparison of compilation with sharing and reference-counting enabled with the uncompiled functional-style UDF dtw and the (optimized) hand-crafted CTE. The definition of dynamic time warping (dtw) and this plot appeared in Chapter 1 before.

- **Batching** turns function f , that processes a single argument list $f(\overline{\text{args}})$, into a function f' that evaluates an array of argument lists $f'([\overline{\text{args}}, \dots, \overline{\text{args}}])$ batched together in a single call (Section 5.6).

Note that tweaks, such as table-valued return types, memoization, and batching, work independently of each other for functional-style UDFs which are compiled either as general-, linear-, and tail-recursive. Templates with all tweaks enabled can be found in Appendix B, where their modular nature is highlighted. All experiments in this chapter were performed with PostgreSQL 13 running on a 64-bit Linux x86 host with 8 Intel Core™ i7 CPUs clocked at 3.66 GHz and 64 GB of RAM, of which 128 MB were dedicated to the database buffer. Timings were averaged over 10 runs, with worst and best runtimes disregarded.


```

1 CREATE FUNCTION dtw(i int, j int)
2 RETURNS real AS $$
3 CASE
4   WHEN i=0 AND j=0 THEN ④0.0
5   WHEN i=0 OR j=0 THEN ⑤∞ -- 'Infinity'::real
6   ELSE (SELECT abs(Z.x - Z.y)
7         +
8         LEAST(①dtw(i-1, j-1),
9              ②dtw(i-1, j ),
10             ③dtw(i , j-1))
11        FROM (X JOIN Y
12              ON ((X.t,Y.t) = (i,j))) AS Z)
13 END;
14 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 5.2: 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.

evaluation					
	in	val	ref	site	fanout
<i>i</i>	<i>i</i> <i>j</i>		<i>i</i> <i>j</i>		
1	(0,0)	0.0	(1,1)	1	3
	(0,1)	∞	(1,1)	2	3
	(1,0)	∞	(1,1)	3	3
	(0,1)	∞	(1,2)	1	3
	(0,2)	∞	(1,2)	2	3
	(1,0)	∞	(2,1)	1	3
	(2,0)	∞	(2,1)	3	3
	2	(0,1)	∞	(1,2)	1
(0,2)		∞	(1,2)	2	3
(1,0)		∞	(2,1)	1	3
(2,0)		∞	(2,1)	3	3
(1,1)		0.0	(2,2)	1	3
(1,1)		0.0	(1,2)	3	3
(1,1)		0.0	(2,1)	2	3
3	(1,1)	0.0	(2,2)	1	3
	(1,2)	1.0	(2,2)	2	3
	(2,1)	0.0	(2,2)	3	3
4	(2,2)	1.0	(2,2)	□	□

Figure 5.3: Table evaluation for dtw(2,2) with reference counting enabled. Note how in each evaluation step *i*, we keep only those rows in the working table that have not yet used to produce a new result.

5.1 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 *previous* iteration [101]. SQL implementations hold these newly found rows of t in the so-called *working table* [97, §7.8], ready to be read by q_2 . In the vanilla compilation (recall Chapter 4), the evaluation CTE of Figure 4.12 keeps both known and new results (see Line 18 in `Traverse`) to ensure that `Schedule` sees all results found so far. Thus, the evaluation of a call graph node may depend on the result of a node found in any lower layer of the bottom-up graph traversal. For example, reusing the dynamic time warp (dtw) from previous Chapter 4, the evaluation of call graph node (1,2) in Figure 4.5a depends on result for nodes (0,2), (0,1), and (1,1) to be present in the working table. As evaluation continues, this leads to monotonically increasing working table sizes containing all these previous (possibly obsolete) results, negatively affecting runtime performance. We find that the *in-degree* of a call graph node determines how often its return value is referenced during the evaluation of parent calls. Thus, the following adaption of the compilation scheme enables *reference counting*, which removes such obsolete results from working tables throughout the bottom-up traversal of the call graph:

1. To the tabular encoding of the results in the evaluation CTE, add utility columns `ref`, `site`, and `fanout`. Together with `in`, these columns represent call graph edges `ref • site → in`. Referee `ref` functionally determines `fanout`. Thus, if the count of rows with the same `ref` is equal to `fanout`, evaluate the result of outgoing call $f(\text{ref})$ and store the newly evaluated results in the working table.
2. Mark those rows used to evaluate $f(\text{ref})$ as referenced and drop them from subsequent working tables.
3. For all newly evaluated results of outgoing call $f(\text{ref})$, we look up their referee's `ref`, `site`, and `fanout` in the call graph and add them to the working table.

Figure 5.3 shows the working tables of each iteration i when evaluating `dtw(2,2)` with

```

1 [WITH RECURSIVE] evaluation(in,val,ref,site,fanout) AS (
2   TABLE base_cases
3   UNION ALL -- recursive UNION ALL
4   (WITH e(in,val,ref,site,fanout) AS (TABLE evaluation),
5     returns(in,val) AS (
6       SELECT go.in,
7         (replace(body(f, [lookup(f, 1), ... , lookup(f, n)]),
8           [(go.in).args])) AS val
9       FROM (SELECT e.ref
10            FROM e
11            GROUP BY e.ref, e.fanout
12            HAVING COUNT(*) = e.fanout
13            ) AS go(in)
14     )
15     SELECT *
16     FROM e
17     WHERE e.fanout IS NOT NULL
18     AND NOT EXISTS (SELECT FROM returns AS r WHERE r.in = e.ref)
19     UNION ALL
20     SELECT r.*, g.in, g.site, g.fanout
21     FROM returns AS r, call_graph AS g
22     WHERE r.in = g.out
23   )
24 )

```

Figure 5.4: Tweaking the template of Figure 4.12 in previous Chapter 4 implements reference counting and enables functional-style UDFs with call sites that reference surrounding row variables.

reference counting enabled. The final step ($i = 4$) lists only the result for (2,2) and, thus, marks the end of the bottom-up traversal and evaluation.

5.1.1 TWEAKING THE SQL TEMPLATE

The vanilla templates are tweaked to implement reference counting. Reference counting also lifts the restriction where recursive call sites were not allowed to reference surrounding row variables as a side effect. Such call sites that reference surrounding row variables we call *correlated*. For correlated call sites, fanout depends on the unknown number of rows the row variable references. Thus, we identify functions with correlated call sites as *n-fold recursive*. Compare this to dtw (see Figure 5.2) which has no correlated call site and which identifies it as 3-fold recursive.

We highlight those changes that lead to the template in Figure 5.4:

RefSchedule Collect referee nodes `ref` whose recursive call results are all present in the working table of evaluation. This decides whether outgoing call $f(\text{ref})$ is ready for evaluation or not. Notice, that custom aggregate `array_gather` is not required anymore, because we look up results in **RefBody** *directly* from the working table.

RefBody For each referee node `in` go, evaluate the body of f with its call sites replaced by subquery:

$$lookup(f, \ell) = \left(\begin{array}{l} \text{SELECT } e.\text{val} \text{ FROM } e \\ \text{WHERE } e.\text{ref} = \text{in} \\ \text{AND } e.\text{site} = \ell \\ \text{AND } e.\text{in} = call(f, \ell) \end{array} \right).$$

The subquery $lookup(f, \ell)$ looks up the result of recursive call `in` of call graph edge `ref •site→in` i.e. the result of `in` at `site` called by `ref`. Recursive call `in` is determined by $call(f, \ell)$ which extracts the call site argument expressions at ℓ . For example: $call(\text{dtw}, \mathbf{1}) = (i-1, j-1)$ (see call site $\mathbf{1}$ in Figure 5.2). Extracting the call site arguments and looking up the results in the working table of evaluation makes the custom aggregate `array_gather` obsolete and, as a side effect, allows compilation of functional-style UDFs with correlated call sites.

RefTraverse Do not keep results in the working table of evaluation that have been used in this evaluation step. For each result of `in` in CTE returns `find` all its referees `ref` in `call_graph` and add the result of `in` at `site` called by `ref` representing call graph edge `ref •site→out`). Indeed, duplicates of the *same* results are added to the working table if more than one node `ref` depends on `in`. For example, recall $(1,1)$ in Figure 4.5a which has its result added three times, because nodes $(2,1)$, $(2,2)$, and $(1,2)$ are its referees. However, this is a small price to pay since, with this tweak, we only join newly discovered results with `call_graph` instead of the entire evaluation CTE. The number of newly discovered results is exactly the number of nodes in the `call_graph`. This improves runtime complexity from $|evaluation| \times |call_graph|$ to $|call_graph|^2$ (recall that $|evaluation| > |call_graph|$). Bottom-up traversal continues until we reach the initial function call, which ends the evaluation and produces the result of f .



```

1 CREATE FUNCTION  $f(\overline{\text{args}})$  RETURNS  $\tau$ 
2 AS $$
3 WITH RECURSIVE call_graph(in,site,fanout,out,val) AS (
4     <see Figure 4.8>
5     ),
6     base_cases(in,val,ref,site,fanout) AS (
7         SELECT g.in, g.val, g_ref.in, g_ref.site,g_ref.fanout
8         FROM call_graph AS g,
9              call_graph AS g_ref
10        WHERE g.fanout = 0
11              AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
12              AND g.in = g_ref.out
13        ),
14     evaluation(in,val,ref,site,fanout) AS (
15         <see Figure 5.4>
16     )
17     SELECT e.val
18     FROM evaluation AS e
19     WHERE e.fanout IS NULL;
20 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 5.5: The compiled SQL code enables reference counting and allows correlated call sites in functional-style UDFs.

In Figure 5.5, regions **RefBase**, **RefEval**, and **RefResult** replace their vanilla counterparts. As before, the `base_cases` CTE initializes the first results at the very bottom layer of the traversal. Indeed, as with **RefTraverse** in CTE evaluation, multiple copies of the same result may be stored in `base_cases` if multiple ancestors depend on it. From table evaluation, **RefResult** extracts f 's return value, which has fanout set to \square , to deliver the function's final result.

Figure 5.6a traces the working table size as CTE evaluation traverses the call graph for `dtw(100,100)`. The vanilla evaluation CTE of Figure 4.12 indeed processes working tables of monotonically increasing size, growing from 201 to 10,201 rows across the 200 iterations. This incurs a noticeable runtime penalty for evaluation processes that recurse deeply (see  in Figures 5.6a and 5.6b). With reference counting, the working table size never exceeds 400 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 runtime (see  in both figures). While sharing helps to keep working table sizes in check during call graph construction, reference counting does the same during evaluation. The evaluation of `dtw(300,300)` builds a working table that never exceeds

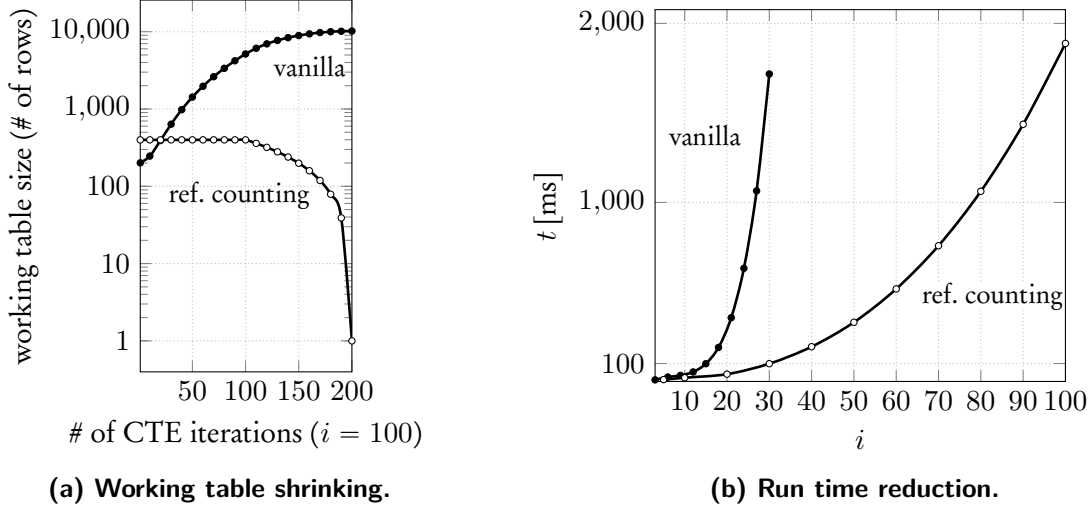


Figure 5.6: Evaluating $\text{dtw}(100,100)$: Impact of reference counting on working table size and CTE runtime.

1200 rows if reference counting is performed. To put into perspective how far we have come, recall Figure 5.1 as it compares dtw between the original UDF (before compilation) and the final compiled UDF.

5.1.2 THE IMPACT OF CALL SHARING

Some functional-style UDFs never share recursive calls throughout call graph construction. Take, for example, function $\text{split}(x, y)$ where x and y represent the boundaries of a range $[x, y]$. split divides this range in the middle and continues to do so recursively until $|x - y| \leq 1$, at which point it returns 1. We define split recursively as follows:

$$\text{split}(x, y) = \begin{cases} 1 & , |x - y| \leq 1 \\ \text{split}(x, x + \frac{|x-y|}{2}) + \text{split}(x + \frac{|x-y|}{2}, y), & \text{otherwise} \end{cases} \quad (\text{split})$$

split will never produce the same recursive call twice during its evaluation. Figure 5.7 shows a direct comparison of split before compilation (\curvearrowright), after compilation without the benefit of call sharing (\curvearrowleft) and, the function implemented as a hand-crafted

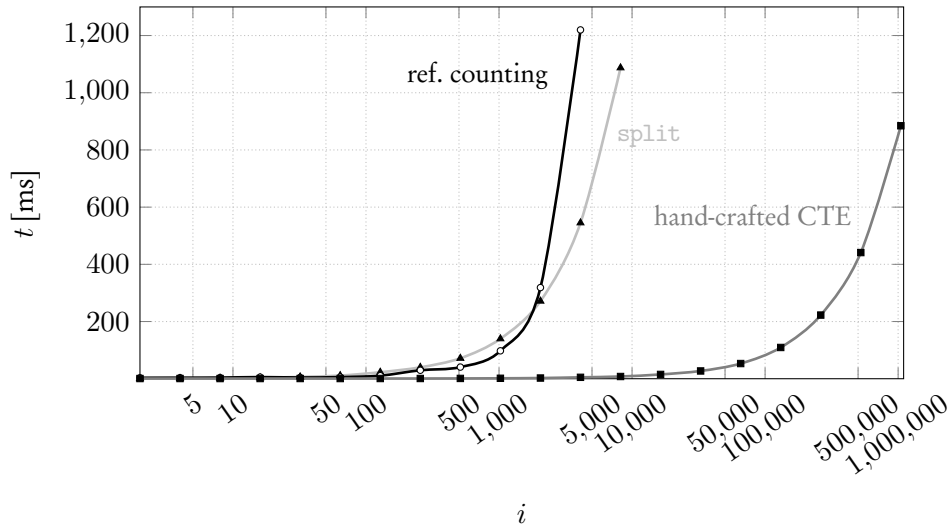


Figure 5.7: Evaluating `split(i, i2)`.

CTE (■). Some functional-style UDFs, whether compiled or not, fall *far* behind when compared to the execution times of a hand-crafted CTE (found in Section C.3). Thus, developers must take into consideration whether their complex computation exhibits call sharing potential before deciding between writing a functional-style UDF (and compiling it) and a hand-crafted implementation using recursive CTEs. In such cases, the developer may be faced with a compromise between readability and performance. Indeed, this underlines that functional-style UDFs are not here to replace recursive CTEs but to give the developer additional options to implement complex computations.

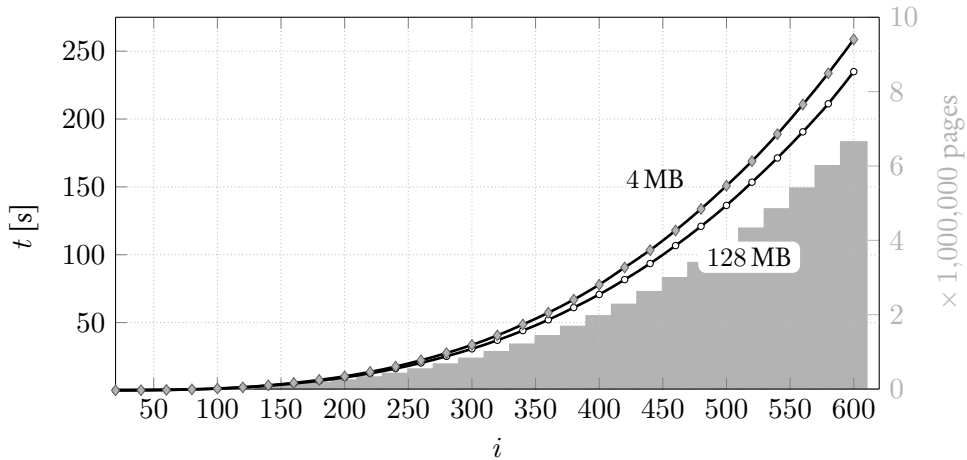





Figure 5.8: Evaluating $\text{dtw}(i, i)$. Compares the runtime difference of the compiled UDF using reference counting with a database buffer size set to 4 MB and 128 MB. Only when the database buffer size is set to 4 MB, do we measure any I/O operations.

5.1.3 RUNTIME EXPERIMENTS

Pushing the approach further, we measure $\text{dtw}(i, i)$ until we reach $i = 600$ exhibiting a call graph with 361,201 nodes. Figure 5.8 reports that the `call_graph` CTE will exceed the PostgreSQL 13 default database buffer size of 4 MB [97] beginning with $\text{dtw}(140, 140)$ (see ). As a consequence, PostgreSQL writes the excess pages of the database buffer to disk and, thus, evaluation CTE must now read from disk when joining with the `call_graph` CTE. The bar chart () plots the throughput of I/O operations measured in pages (the size of a single page is set to the PostgreSQL default: 8192 byte [97]). Increasing the database buffer size to a modest 128 MB (*i.e.*, 0.2% of the host’s RAM of 64 GB), not a single buffer read or write I/O operation is performed by PostgreSQL (see ). For example: $\text{dtw}(600, 600)$ with a database buffer size of 4 MB requires 6,671,512 I/O operations (of which only 99.7% are read operations). Increasing the database buffer size to 128 MB then improves its runtimes from 258.7 seconds to 234.9 seconds, shaving off almost 10% of the time needed to produce the result with minimal effort. In other words, the I/O operations cost us 23.8 seconds extra unless the buffer size *is moderately increased* to fit the `call_graph`.

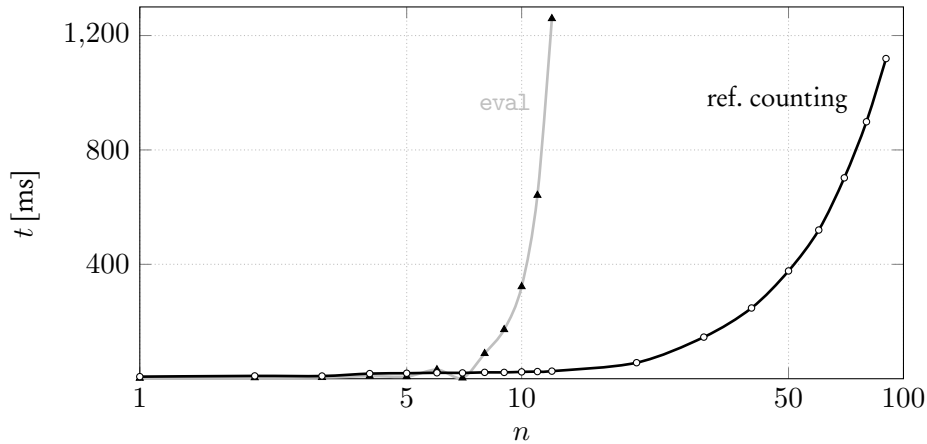


Figure 5.9: Evaluating $\text{eval}(e)$ where e is an expression which evaluates 2^n binary operators.

The rest of this section compares other functional-style UDF use cases:

eval: Function $\text{eval}(e_k)$ evaluates an arithmetic expression $e_k = (op, e_l, e_r, v)$.

- Each expression e_i is uniquely identified by integer i ,
- op can either be a binary operator $(+, -, \times, \div)$ or ℓ , and
- v can be any real number.

We assume that the same expressions also have the same identifier k . Function eval is recursively defined as:

$$\begin{aligned}
 \text{eval}((\ell, e_l, e_r, v)) &= v \\
 \text{eval}((+, e_l, e_r, v)) &= \text{eval}(e_l) + \text{eval}(e_r) \\
 \text{eval}((- , e_l, e_r, v)) &= \text{eval}(e_l) - \text{eval}(e_r) \\
 \text{eval}((\times, e_l, e_r, v)) &= \text{eval}(e_l) \times \text{eval}(e_r) \\
 \text{eval}((\div, e_l, e_r, v)) &= \text{eval}(e_l) \div \text{eval}(e_r) .
 \end{aligned}
 \tag{eval}$$

Function eval exhibits 2-fold recursion. Compilation enables sharing of common subexpressions and improves runtimes by order of magnitude compared to eval before compilation (see Figure 5.9). Compilation also turns the original top-down expression interpreter into a bottom-up variant that processes all independent subexpressions step by step in the evaluation CTE.

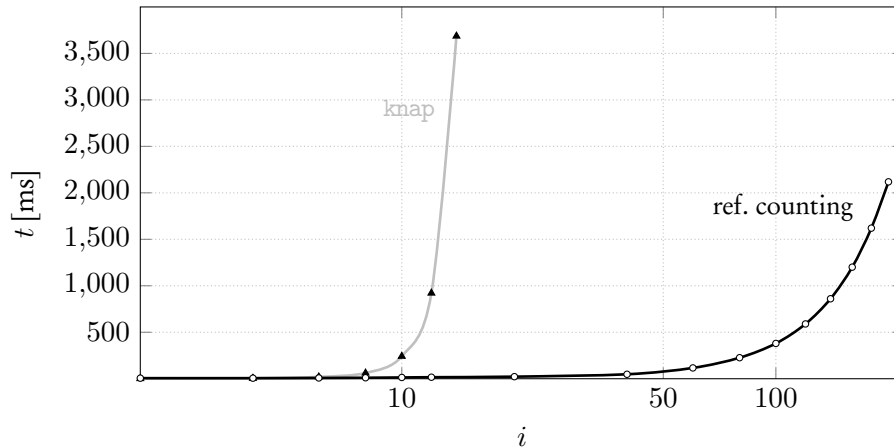


Figure 5.10: Evaluating $\text{knap}(i, i)$.

knap: Function $\text{knap}(k, u)$ implements an algorithm to solve the *0-1 Knapsack* problem [88]. Given items $i \in 1, \dots, n$ where each item has weight w_i and value p_i . Then $\text{knap}(k, u)$ maximizes the sum of values of k items that fit into a knapsack with carrying capacity u . Function knap is recursively defined as:

$$\begin{aligned} \text{knap}(1, u) &= 0 \\ \text{knap}(k, u) &= \begin{cases} \text{knap}(k-1, u) & , w_k > u \\ \max \left\{ \begin{array}{l} \text{knap}(k-1, u) \\ \text{knap}(k-1, u-w_k) + p_k \end{array} \right\}, & \text{otherwise} \end{cases} \end{aligned} \quad (\text{knap})$$

Function knap exhibits 2-fold recursion. The *0-1 Knapsack* algorithm is used as the running example in Chapter 2 to highlight one of the functions developers would prefer to implement in functional-style. Figure 5.10 reports that, if a developer then also compiles knap , the runtime improves significantly. Call sharing turns the brute force approach into one that utilizes dynamic programming, where redundant subproblems are evaluated once instead of many times. Thus, for this use case, developers can expect both: readability *and* runtime improvements.

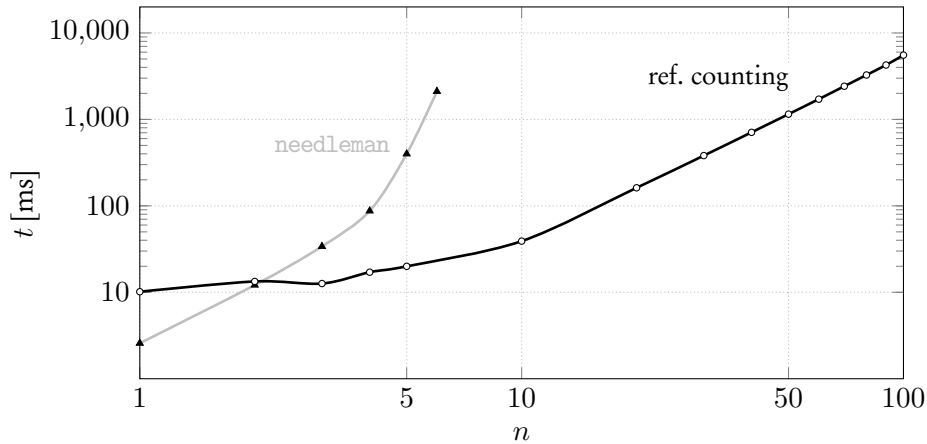


Figure 5.11: Evaluating $\text{needleman}(p, q, n, n, g)$ compares two protein sequences p and q of length n with gap penalty $g = -5$.

needleman: Function $\text{needleman}(p, q, n, n, g)$ implements the Needleman-Wunsch algorithm [91] which compares two protein sequences p and q of length n and rates them based on gap penalty g and a two-dimensional scoring matrix M (here: BLOSUM62 [99]). Notation s_i accesses a single amino acid in protein sequence s at position i . The scoring matrix M takes two single amino acids a and b and produces a score (in short: $M[a][b]$). Function needleman is a well-known algorithm in Bioinformatics and is recursively defined as:

$$\begin{aligned}
 \text{needleman}(p, q, 0, 0, g) &= 0 \\
 \text{needleman}(p, q, i, 0, g) &= i \times g \\
 \text{needleman}(p, q, 0, j, g) &= j \times g \\
 \text{needleman}(p, q, i, j, g) &= \max \left\{ \begin{array}{l} \text{needleman}(p, q, i-1, j, g) + g \\ \text{needleman}(p, q, i, j-1, g) + g \\ \text{needleman}(p, q, i-1, j-1, g) + M[p_i][q_j] \end{array} \right\}.
 \end{aligned}
 \tag{needleman}$$

Function needleman exhibits 3-fold recursion. Figure 5.11 reports on the runtime of comparing two sequences p and q of equal length n . The call graph is structured almost identically to dtw and, thus, compiling improves its exponential growth to $O(n^2)$ through call sharing.

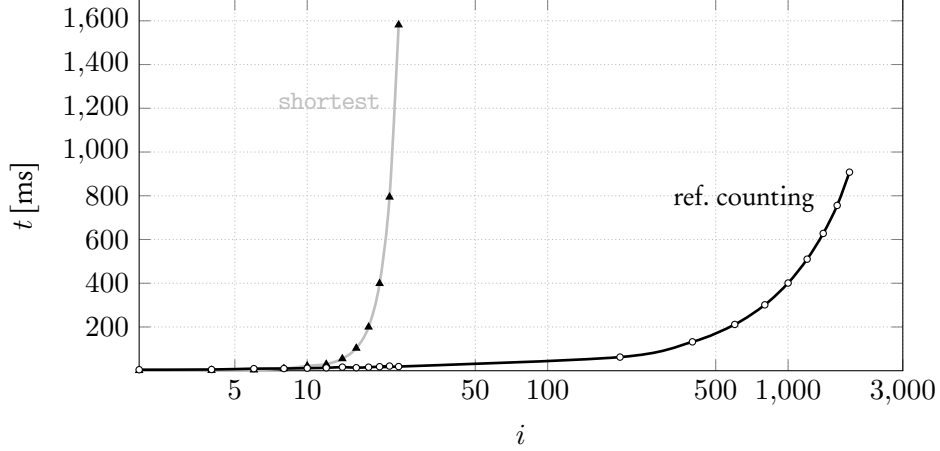


Figure 5.12: Evaluating $\text{shortest}(n_1, n_2)$ where n_1 and n_2 are nodes where finding the shortest path between them involves visiting exactly i nodes. Average fanout of 1.5.

shortest: Given a graph with n nodes, where $w_{i \rightarrow j}$ denotes the weight of an edge $i \rightarrow j$ between two nodes v_i and v_j with $i, j \in 1, \dots, n$. Function $\text{shortest}(i, j)$ finds the shortest path length between two nodes v_i and v_j and returns ∞ if there is none. Function shortest is recursively defined as:

$$\begin{aligned} \text{shortest}(i, i) &= 0 \\ \text{shortest}(i, j) &= \min(\{w_{i \rightarrow k} + \text{shortest}(k, j) \mid \forall i \rightarrow k\} \cup \{\infty\}) . \end{aligned} \quad (\text{shortest})$$

shortest call site is correlated and, thus, exhibits n -fold recursion. Before compilation and without call sharing, any edge between two nodes may be visited many times. Call sharing prevents this; thus, any edge is traversed exactly once. Figure 5.12 reports that execution runtime improves significantly after compilation.

Functional-style UDFs `dtw` and `split`, as well as their compiled and manually crafted counterparts, can be found in Appendix C. Other use cases can be found in Appendix D. Among them are queries on mathematical problems (Section D.1), more queries over graphs (Section D.2, Section D.4) and string processing queries (Section D.3).

5.1.4 SUMMARY

The tweaks we describe in this section are a general improvement over the vanilla templates introduced in the previous Chapter 4. Introducing reference counting improves execution times and, as a side effect, enables compilation of functional-style UDFs which exhibit correlated call sites. However, functions such as `split` highlight that not all functional-style UDFs have improved execution times when compiled. Generally speaking: compilation leads to better performance for general recursive functions, depending on how much the call graph can be shrunk by call sharing. In later Sections 5.3 to 5.6, we describe other methods (besides call sharing) that allow the compiler to exploit the presence of a call graph in the evaluation process. Note that all tweaks in the following sections use this improved template as their foundation.

```

1 WITH RECURSIVE call_graph(in, site, fanout, out, vals, "empty?",rid) AS (
2   SELECT ROW(f.args), NULL::int, NULL::bigint, ROW(f.args), NULL, false, NULL::bigint
3   UNION -- recursive UNION Invoke
4   SELECT g.out, edges.*
5   FROM call_graph AS g,
6   LATERAL (
7     WITH slices(site, out) AS (
8       SELECT 1 AS site, out FROM (replace(slice(f, s1), [(g.out).args])) AS _(out) Slices
9       UNION
10      :
11      UNION
12      SELECT n AS site, out FROM (replace(slice(f, sn), [(g.out).args])) AS _(out)
13     ),
14     calls(site, fanout, out, vals, "empty?",rid) AS (
15       SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL, false, NULL::bigint
16       FROM slices AS s
17       WHERE s.out <> ⊥ Calls
18     ),
19     values(vals) AS (
20       replace(body(f, [(VALUES NULL), ..., (VALUES NULL)]), [(g.out).args]) TblValues
21     )
22     TABLE calls
23     UNION ALL
24     SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out, v.vals, false,
25     ROW_NUMBER() OVER ()
26     FROM values AS v
27     WHERE NOT EXISTS (TABLE calls)
28     UNION ALL
29     SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out, NULL, true, 1
30     WHERE NOT EXISTS (TABLE calls)
31     AND NOT EXISTS (TABLE values) TblConstruct
32   ) AS edges(site, fanout, out, vals, "empty?", rid)
33   WHERE g.fanout > 0 OR g.fanout IS NULL
34 )

```

Figure 5.13: Tweaking the template of Figure 4.8 in Chapter 4 enables functional-style UDFs with table-valued return types.

```

1 [WITH RECURSIVE] evaluation(in,vals,"empty?",rid,ref,fanout) AS (
2 TABLE base_cases
3 UNION ALL -- recursive UNION ALL
4 (WITH e(in,vals,"empty?",rid,ref,fanout) AS (TABLE evaluation),
5 returns(in,vals,"empty?",rid) AS (
6     SELECT go.in, result.*
7     FROM (SELECT e.ref
8           FROM e
9           WHERE e.rid = 1
10          GROUP BY e.ref, e.fanout
11          HAVING COUNT(*) = e.fanout
12         ) AS go(in),
13     LATERAL (
14         WITH result(vals) AS (
15             replace(body(f, [lookup_Tbl(f, 1), ..., lookup_Tbl(f, n)]),
16                 [(go.in).args])
17         )
18         SELECT r.*, false, ROW_NUMBER() OVER () FROM result AS r
19         UNION ALL
20         SELECT NULL, true, 1 WHERE NOT EXISTS (TABLE result)
21         ) AS result(vals,"empty?")
22     )
23     SELECT *
24     FROM e
25     WHERE e.fanout IS NOT NULL
26     AND NOT EXISTS (SELECT FROM returns AS r WHERE r.in = e.ref)
27     UNION ALL
28     SELECT r.*, g.in, g.fanout
29     FROM returns AS r, call_graph AS g
30     WHERE r.in = g.out
31 )
32 )

```

TblBody

TblSchedule

RefTraverse

Figure 5.14: Tweaking the template of Figure 5.4 enables compiling functional-style UDFs with table-valued return types.

```

1 CREATE FUNCTION f( $\overline{\text{args}}$ ) RETURNS TABLE( $\overline{\text{vals}}$ )
2 AS $$
3 WITH RECURSIVE call_graph(in,site,out, $\overline{\text{vals}}$ , "empty?",rid) AS (
4     <see Figure 5.13>
5 ),
6 base_cases(in, $\overline{\text{vals}}$ , "empty?",rid,ref,fanout) AS (
7     SELECT g.in, g.val, g."empty?", g.rid, g_ref.in, g_ref.fanout
8     FROM call_graph AS g,
9          call_graph AS g_ref
10    WHERE g.fanout = 0
11          AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
12          AND g.in = g_ref.out
13 ),
14 evaluation(in, $\overline{\text{vals}}$ , "empty?",rid,ref,fanout) AS (
15     <see Figure 5.14>
16 )
17 SELECT (e. $\overline{\text{vals}}$ ).*
18 FROM evaluation AS e
19 WHERE e.fanout IS NULL
20 AND NOT e."empty?";
21 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 5.15: The compiled SQL code with support for table-valued return types TABLE($\overline{\text{vals}}$).

5.2 TABLE-VALUED FUNCTIONS

Tweaking the template in Section 5.1 further enables functional-style UDFs with table-valued return types TABLE($\overline{\text{vals}}$) i.e., enable the recursive CTEs `call_graph` and `evaluation` to accommodate for return values that have multiple rows. The general idea here is to return all value columns `val` in those recursive CTEs into a table-valued context. Furthermore, the case where an empty table is returned must now be considered.

5.2.1 TWEAKING THE SQL TEMPLATE

The changes to the templates are highlighted with regions in Figures 5.13 to 5.15. We highlight some in detail:

TblValues Base cases are now stored inside values CTE instead of being inlined inside the SELECT-clause directly. The values CTE is defined through:
`replace(body(f, [(VALUES $\overline{\text{NULL}}$), ... , (VALUES $\overline{\text{NULL}}$)]), [(g.out).args])` First, the function body of f is passed to `body` which replaces all call sites $f(\text{args})$ with `(VALUES $\overline{\text{NULL}}$)`,

where $\overline{\text{NULL}}$ expands to a row (NULL, ..., NULL) with the same number of columns as table-valued return type $\text{TABLE}(\overline{\text{vals}})$. Then, *replace* replaces each of f 's arguments $\overline{\text{args}}$ in f 's body with $[(g.out).args]$.

TblConstruct Window function $\text{ROW_NUMBER}() \text{ OVER } ()$ [97] adds a *row id* rid for each row in values to prevent the recursive UNION of call_graph from pruning duplicate rows. We add a third case where column "empty?" is TRUE, only if no new recursive call sites in calls nor base cases in values are present (see lines 29-31 in Figure 5.13).

TblSchedule Collect referee nodes ref whose recursive call results are all present in the working table of evaluation. Indeed, it is sufficient to only check for rows with $\text{rid} = 1$.

TblBody Similar to $\text{lookup}(f, \ell)$ in Section 5.1.1, $\text{lookup}_{\text{Tbl}}(f, \ell)$ looks up the result of recursive call in of call graph edge $\text{ref} \cdot \text{site} \rightarrow \text{in}$ i.e., the result of recursive call in at site called from ref :

$$\text{lookup}_{\text{Tbl}}(a) = \left(\begin{array}{l} \text{SELECT } e.\overline{\text{vals}} \text{ FROM } e \\ \text{WHERE } e.\text{ref} = \text{go.in} \\ \text{AND } e.\text{site} = \ell \\ \text{AND } e.\text{in} = \text{call}(f, \ell) \\ \text{AND } \text{NOT } e.\text{"empty?"} \end{array} \right) .$$

$\text{lookup}_{\text{Tbl}}$ adds predicate $\text{NOT } e.\text{"empty?"}$ to ensure that the subquery returns no rows, if the result of recursive call site $\text{call}(f, \ell)$ is an empty table.

5.2.2 RUNTIME EXPERIMENTS

When compiling and evaluating functions with table-valued return types, we can expect the working table of the evaluation CTE to be much larger compared to functions with scalar return types. Take, for example, function $\text{bom}(p_b)$, which returns the table of all subparts of p_b i.e., the bill of materials. The materials are stored in table $M(p, s, q)$, which holds all parts p and its subparts s with quantity q . Function bom , which returns a table with schema (p_r, s_r, q_r) , is recursively defined using well known

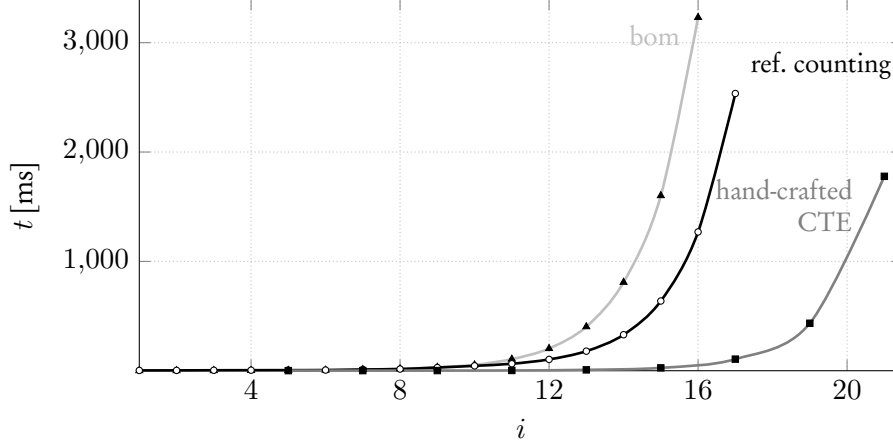


Figure 5.16: Evaluating table-valued $\text{bom}(p)$, where p is a part made out of i (nested) subparts in total.

relational algebra operators σ (selection), π (projection), \cup (union) and \bowtie (join):

$$\text{bom}(p_b) = \sigma_{p=p_c}(M) \cup \pi_{p_r, s_r, q \times q_r} \left(\bigcup_{(p, s, q) \in M} (\{ (p, s, q) \} \bowtie_{p=pr} \text{bom}(s)) \right). \quad (\text{bom})$$

Figure 5.16 reports a runtime improvement not as drastic as with compiled functions with scalar return types. Indeed, despite having call sharing and reference counting enabled, a single call result in the working table of CTE evaluation may still encode a large table (possibly multiple times, due to reference counting). Appendix C shows the implementation of functional-style UDF bom , as well as its compiled and manually crafted counterparts.

5.2.3 SUMMARY

We show that we support compilation of functions with table-valued return types by tweaking the template introduced in Section 5.1. However, there is a price to pay which leads to a not-so-drastic runtime improvement when compiled. If runtimes are a high priority for the developer and the decline of readability acceptable, writing hand-crafted recursive CTEs for table-valued functions may be the preferred option. Note that tail-recursive functions with table-valued return types do not face this problem, as we will see later in Section 5.4.

```

1  [WITH RECURSIVE] evaluation(in,val,ref,site,fanout) AS (
2  TABLE base_cases
3  UNION ALL -- recursive UNION ALL
4  (WITH e(in,val,ref,site,fanout) AS (TABLE evaluation),
5   returns(in,val) AS (
6     SELECT go.in,
7           (replace(body(f, [lookup(f, 1), ..., lookup(f, n)]),
8            [(go.in).args ])) AS val
9   FROM (SELECT e.ref
10        FROM e
11        GROUP BY e.ref, e.fanout
12        HAVING COUNT(*) = e.fanout
13        ) AS go(in)
14  )
15  SELECT r.*, g.in, g.site, g.fanout
16  FROM returns AS r, call_graph AS g
17  WHERE r.in = g.out
18  )
19  )

```

Figure 5.17: This template for evaluation used for linear recursive functions is based on the template found in Figure 5.4.

5.3 LINEAR RECURSION

Linear recursive functions [55] exhibit common recursion patterns that allow the compiler to assume a certain call graph structure which simplifies bottom-up traversal CTE evaluation. Such functions f can be characterized by their (prefix tree of) evaluation paths (recall Section 4.6 and more specifically Figure 4.16). Specifically, function f is *linear recursive* if each subtree of paths rooted in a control flow label $\textcircled{0}$ contains *at most* one recursive call site. In the grammar of Chapter 3, dark control flow labels \bullet are associated with CASE expressions. 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 have to keep older results and thus no reference counting is required i.e. precisely one node will be ready for evaluation in each iteration of bottom-up traversal.

5.3.1 TWEAKING THE SQL TEMPLATE

The compiler exploits linear recursion by simplifying the SQL template for evaluation (see Figure 5.17). As the evaluation process walks the chain back to the root node, we

```

1 CREATE FUNCTION f(args) RETURNS  $\tau$ 
2 AS $$
3 WITH RECURSIVE call_graph(in,site,fanout,out,val) AS (
4     <see Figure 4.8>
5     ),
6     base_cases(in,val,ref,site,fanout) AS (
7         SELECT g.in, g.val, g_ref.in, g_ref.site,g_ref.fanout
8         FROM call_graph AS g,
9              call_graph AS g_ref
10        WHERE g.fanout = 0
11              AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
12              AND g.in = g_ref.out
13        ),
14     evaluation(in,val,ref,site,fanout) AS (
15         <see Figure 5.17>
16     )
17     SELECT e.val
18     FROM evaluation AS e
19     WHERE e.fanout IS NULL;
20 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 5.18: This SQL code is emitted, whenever the compiler detects linear recursion.

only keep the most recent result in the working table discarding all that came before (see **LinTraverse**). Note that the linear recursion tweaks can be applied independently of the other compiler tweaks presented in this chapter. Figure 5.18 show the SQL template for linear recursive functions which the compiler emits. Templates with all tweaks enabled can be found in Appendix B, where their modular nature is highlighted.

5.3.2 RUNTIME EXPERIMENTS

Function $\text{paths}(d)$ reconstructs the path of a directory d in a filesystem [87]. A directory d has a name n_d and a parent directory p_d . If d is the root directory, then its parent directory p_d is NULL. Function paths is recursively defined using string concatenation $||$ as:

$$\text{paths}(d) = \begin{cases} '/', & \text{if } p_d \text{ is NULL} \\ \text{paths}(p_d) || n_d || '/', & \text{otherwise} \end{cases} \quad (\text{paths})$$

Function paths is linear recursive, which the compiler detects. In Figure 5.19, we compare function paths before compilation (paths) with its compiled counterpart

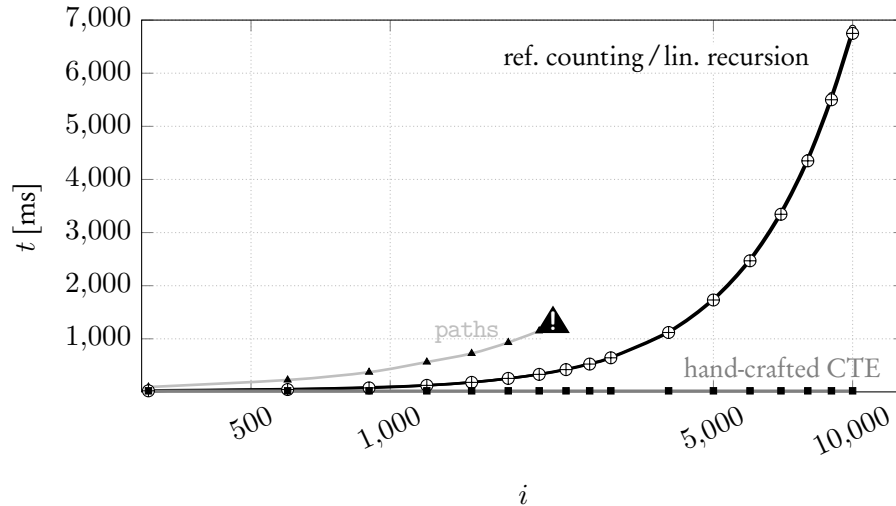


Figure 5.19: Evaluating $\text{paths}(d)$ reconstructs the path of a directory d nested inside i parent directories in a file system. With increasing i , evaluation of fsm pre-compilation (\blacktriangle) terminates prematurely due to stack overflow (\blacktriangle).

with (\oplus) and without (\circ) linear recursion enabled. Observing paths before compilation with increasing directory nesting i quickly reaches the stack size limit at $i = 2400$ and terminates prematurely (marked with \blacktriangle). Compilation removes any recursive calls and thus the stack size limit. However, linear recursion does not save us from joining each new result with call_graph which prevents them from reaching linear complexity where it would be expected. Indeed, Figure 5.19 reports that linear recursive optimizations do not further improve the execution times, which remain far above the execution times of carefully hand-crafted CTEs (\blacksquare). Recursive CTEs allow implementations of linear recursions to exhibit linear complexity, thus making them a great match for traversing hierarchical data structures in a linear fashion in terms of runtime performance. Another linear recursive use case can be found in Section D.5 which implements a finite state machine.

5.3.3 SUMMARY

The characteristic call chain of a linear-recursive function puts the spotlight on the stack size limit imposed on recursive functions by the underlying database engine and

```

1 CREATE FUNCTION  $f(\overline{\text{args}})$  RETURNS  $\tau$ 
2 AS $$
3 WITH { ITERATE }
4      { RECURSIVE } call_graph(in,site,fanout,out,val) AS (
5      <see Figure 4.8>
6      )
7 SELECT g.val
8 FROM call_graph AS g
9 WHERE g.fanout = 0;
10 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 5.20: This SQL code is emitted, whenever the compiler detects tail recursion.

the operating system. Indeed, compiling the function removes any recursive call sites, and thus the function evaluates calls that would typically exceed the recursion depth limit. Detecting linear recursion also allows the compiler to apply minimal tweaks to the SQL template. However, the execution times improvements are only marginal. In Section 5.4, we build upon linear recursion optimization and improve execution times significantly whenever the compiler detects a *tail-recursive* function.

5.4 TAIL RECURSION

Function f is *tail-recursive*, if f is linear recursive and all recursive call sites *immediately* follow their control flow label without other labels in between. A tail-recursive function f does not perform any computation after it returns from its (one) recursive tail call [103]. Instead, computation is performed in accumulating function arguments. The accumulators form the final result once the function reaches its base case.

5.4.1 TWEAKING THE SQL TEMPLATE

When call graph construction is complete, the final return value of f is already known by the only base case edge. Bottom-up graph traversal is not required, and we can immediately read the result off of the base case edge. A separate evaluation step `base_case` and `evaluation` are not required here. Thus, a much more compact template develops which the compiler emits when processing tail-recursive functions (see Figure 5.20). Note how `TailResult` extracts the return value from the node with a fanout of 0 as soon as call graph construction concludes. As before, this tail recursion tweak can be

applied independently of other compiler tweaks presented in this chapter. Templates with all tweaks enabled can be found in Appendix B, where their modular nature is highlighted.

5.4.2 RUNTIME EXPERIMENTS

Compiling tail-recursive functions removes the need to traverse the call graph bottom-up. Thus, we have to CTE scan `call_graph` only once when looking up the final result, which removes the bottleneck that kept compiled functions from ever reaching linear runtime complexity.

`vm` Take, for example, function `vm` previously introduced in Chapter 1. Function `vm(Pi, R)` takes an initial register state `R` and a program instruction `Pi` at position $i > 0$ of program `P` and computes its result. Single registers in `R` at position j can be accessed with `R[j]`. Binary operator `R[t] ← e` replaces the value of the single register `t` in `R` with the result of expression `e` and returns the modified registers. Function `vm` is recursively defined as:

$$\begin{aligned}
 \text{vm}((i, \text{hlt}, s), R) &= R[s] \\
 \text{vm}((i, \text{lod}, t, x), R) &= \text{vm}(P_{i+1}, R[t] \leftarrow x) \\
 \text{vm}((i, \text{mov}, t, s), R) &= \text{vm}(P_{i+1}, R[t] \leftarrow R[s]) \\
 \text{vm}((i, \text{add}, t, s_1, s_2), R) &= \text{vm}(P_{i+1}, R[t] \leftarrow R[s_1] + R[s_2]) \\
 \text{vm}((i, \text{mul}, t, s_1, s_2), R) &= \text{vm}(P_{i+1}, R[t] \leftarrow R[s_1] * R[s_2]) \\
 \text{vm}((i, \text{div}, t, s_1, s_2), R) &= \text{vm}(P_{i+1}, R[t] \leftarrow R[s_1] / R[s_2]) \\
 \text{vm}((i, \text{mod}, t, s_1, s_2), R) &= \text{vm}(P_{i+1}, R[t] \leftarrow R[s_1] \% R[s_2]) \\
 \text{vm}((i, \text{jmp}, a), R) &= \text{vm}(P_a, R) \\
 \text{vm}((i, \text{jeq}, t, s, a), R) &= \begin{cases} \text{vm}(P_a, R) & , R[s] = R[t] \\ \text{vm}(P_{i+1}, R) & , \text{otherwise} \end{cases}
 \end{aligned} \tag{vm}$$

In Figure 5.21, we compare function `vm` before (↖↗) and after compilation which is separated depending on whether tail-recursive optimization is enabled (⊗⊗) or not (↖↗). Comparing `vm` with and without tail recursion optimization highlights the benefits of such optimizations. Indeed, removing the need to traverse the call graph pushes the runtime complexity for `vm` into linear range $O(n)$, which further

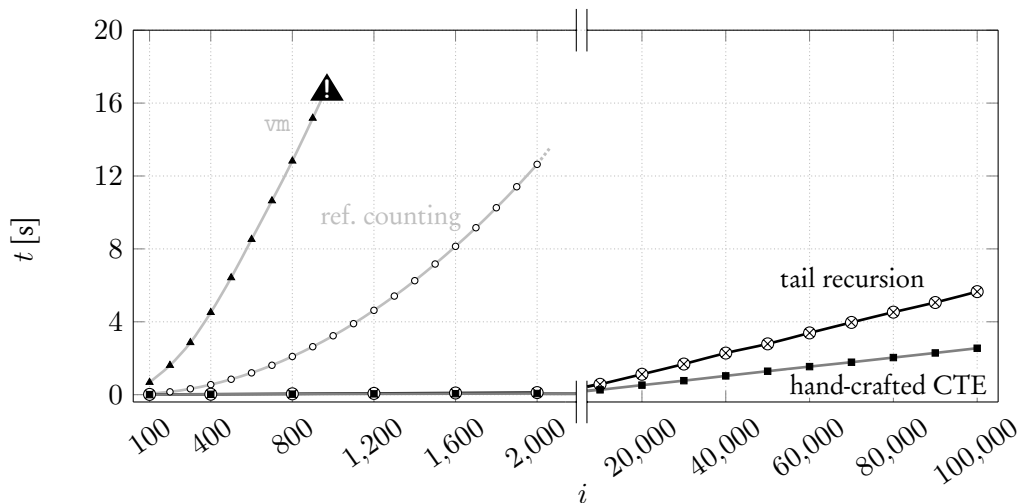


Figure 5.21: Evaluate $vm(r)$, where r is the initial register state which repeats a loop i times. With increasing i , evaluation of vm pre-compilation (\blacktriangle) terminates prematurely due to stack overflow (\blacktriangle).

closes the gap between compilation and carefully hand-crafted implementation. Functional-style UDF vm and its compiled and manually crafted counterparts can be found in Appendix C.

parcels Function $parcels(s)$ begins at parcel station s and visits all subsequent parcel stations until it reaches one which stores items I_s and returns them. A parcel station s points to precisely one other parcel station n_s . The parcel stations are always arranged so that following them one after another leads to a target station containing items. Function $has_items(s)$ simply returns TRUE, if station s has items. Function $parcels$ is recursively defined as:

$$\begin{aligned} \text{parcels}(s, \text{TRUE}) &= I_s \\ \text{parcels}(s, \text{FALSE}) &= \text{parcels}(n_s, \text{has_items}(n_s)) \end{aligned} \quad (\text{parcels})$$

The functional-style implementation of this function expects to return many rows. Figure 5.22 reports that tail-recursive functions improve runtime performance even when expecting table-valued return types. Indeed, this results from not having to traverse the call graph bottom-up.

Other use cases can be found in Appendix D. Among them are queries over geometric

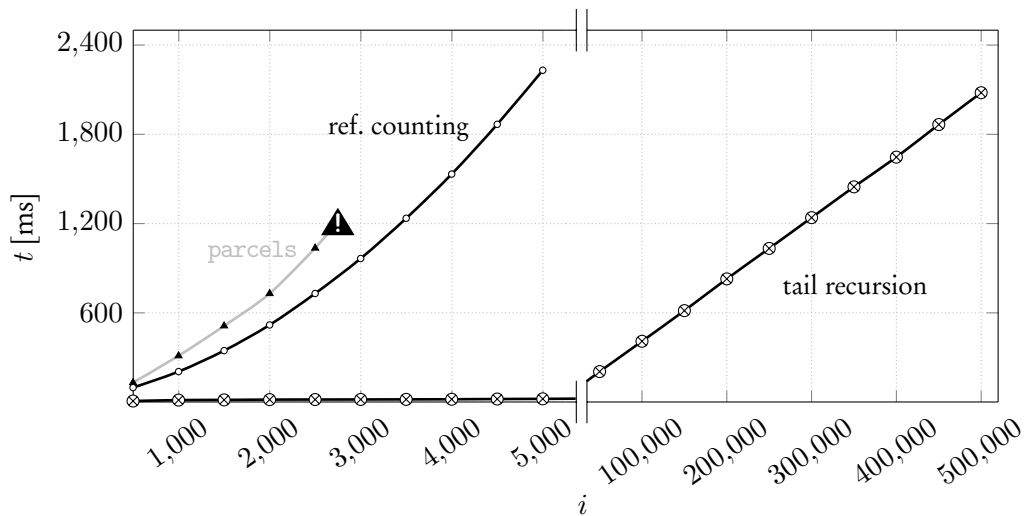


Figure 5.22: Evaluating `parcels(s)`. Starting at station s , `parcels` visits i other stations before it finds all parcels ready and returns them.

shapes (Section D.6) and queries that generate fractals (Section D.7).

5.4.3 WITH ITERATE

Beyond runtime savings, tail recursion promises to be space-efficient as “*tail recursion needs no stack.*”. PostgreSQL fails to exploit this potential when it executes non-compiled functional-style UDFs. We describe a way to allow the developer to exploit this fact for tail-recursive functions by adding the optional tag `ITERATE` (see Chapter 3). 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 5.20 only extracts its single base case edge. Keeping the most recently generated row in table `call_graph` thus would suffice. This is precisely the behavior of the hypothetical `WITH ITERATE` construct [72]. Adding the construct to PostgreSQL 13 amounts to a modest local change. If the developer enables this tail-recursive exclusive optimization, `WITH ITERATE` replaces `WITH RECURSIVE` in the template of Figure 5.20. The system then allocates only a single-row working table during the entire function evaluation process. Figure 5.23 reports, however, that

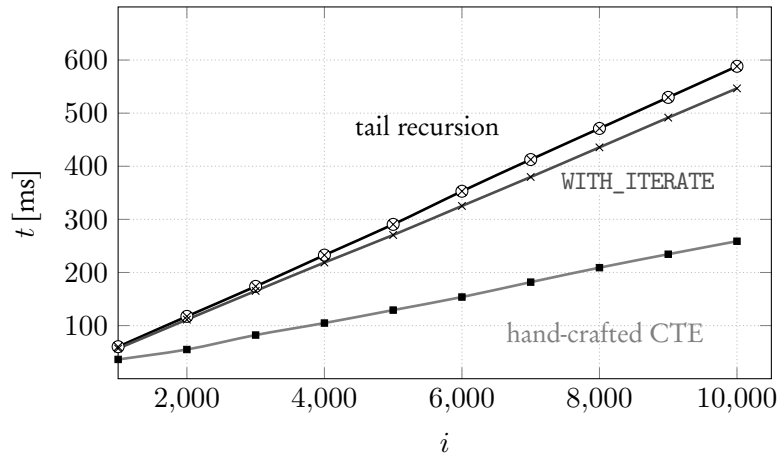


Figure 5.23: Evaluate $\text{vm}(r)$, where r is the initial register state which repeats a loop i times.

the time-saving improvements of WITH ITERATE (\times) are only slight when compared to WITH RECURSIVE (\circ) and do not push runtimes significantly closer to that of the carefully hand-crafted recursive CTE (\blacksquare). Thus, we conclude that WITH ITERATE, despite its slight time-saving improvements, is not something we see as a must-have feature when compiling tail-recursive functions.

5.4.4 SUMMARY

Detecting a tail-recursive function allows the compiler to target a template that has no need for bottom-up traversal of the call graph. Constructing the call graph and returning the result is all that has to be done for tail-recursive functions. Indeed, this improves the lower bound of the runtime complexity to be linear. Thus, if the original function runtime characteristic is linear, then this holds for compiled function as well which.

5.5 MEMOIZATION

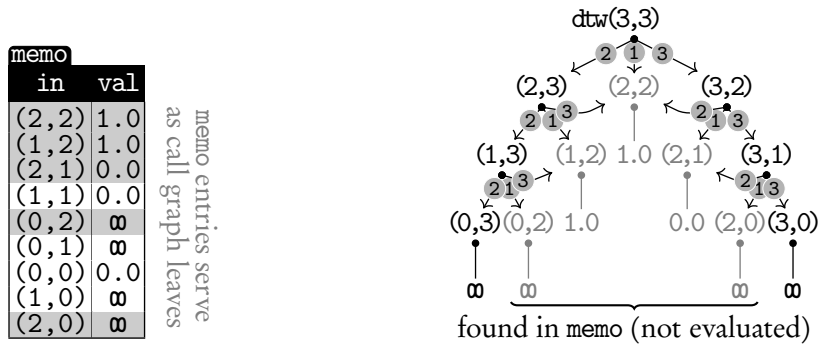
Evaluating a compiled UDF f is a two-phase approach where we first construct the call graph top to bottom by collecting all necessary calls on the way down and then evaluate the call graph bottom-up. Thus, we can assume that on the way up, each call in the call graph will have its results stored in the evaluation CTE. So far, the compiled UDF f extracts exactly the result associated with the call graph's root $\overline{\text{args}}$ (see **RefResult** in Figure 5.5), returns it and then drops all results. 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 [82], 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 [97, §38.7].

Entries in table `evaluation` can be used to accelerate the evaluation of future calls to f . To implement this style of *memoization* [89] for compiled function 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 $f(\overline{\text{args}})$, treat the entries found in `memo` like additional *base cases*.

In the call graph construction, each such extra base case edge `in \rightarrow val` replaces an entire subgraph (with root `in`) whose recursive calls need not be evaluated since `val` is already available. Figure 5.24b shows the call graph for `dtw(3,3)` which has been constructed based on the return values of an earlier `dtw(2,2)` invocation (See Figure 5.24a). In this case, only the 7 calls at the fringes of the graph for `dtw(3,3)` remain to be evaluated (down from 16 calls without memoization). Note that this particular flavor of memoization is already beneficial if the `memo` table holds the root(s) of *any subgraph* of the current call graph [62]. In a sequence of invocations of f , we can thus expect to start saving evaluation effort early on. This is in contrast to plain memoization, which only remembers the single return value at *the root* of an evaluated call graph [94].



(a) memo after dtw(2,2). (b) Memoization prunes the call graph for dtw(3,3).

Figure 5.24: With memoization enabled: evaluate dtw(2,2), then dtw(3,3).

5.5.1 TWEAKING THE SQL TEMPLATE

Figures 5.25 and 5.26 present the SQL templates used when memoization is enabled which are based on the template introduced in Section 5.1. We highlight the necessary changes:

MemLookup If the current recursive call can be found in the memo table, return its result and treat it as a base case later in **MemConstruct**. Otherwise, continue constructing the call graph as usual.

MemStore Once evaluation concludes, add all (intermediate) results (in, val) found in evaluation to table memo. The clause ON CONFLICT DO NOTHING [97] silently discards any results already stored in memo to avoid duplicate entries.

Note that the memoization tweaks can be applied independently of other (optional) compiler tweaks presented in this chapter. Templates with all tweaks enabled can be found in Appendix B, where their modular nature is highlighted. Furthermore, tail-recursive optimization (recall Section 5.4) does not traverse the call graph bottom-up, thus, memoization on top of tail recursion only stores the root result but may still profit from these results in the call graph construction phase.

```

1 WITH RECURSIVE call_graph(in, site, fanout, out, val) AS (
2   SELECT ROW( $\overline{f.args}$ ) AS in, NULL::int, NULL::bigint, ROW( $\overline{f.args}$ ), NULL:: $\tau$ 
3   UNION -- recursive UNION Invoke
4   SELECT g.out, edges.*
5   FROM call_graph AS g,
6   LATERAL (
7     WITH memoization(site, fanout, out, val) AS (
8       SELECT NULL::int, 0, m.in, m.val
9       FROM memo AS m
10      WHERE g.out = m.in MemLookup
11     ),
12     slices(site, out) AS ( Slices
13       SELECT 1 AS site, out FROM (replace(slice( $f, s_1$ ), [(g.out). $\overline{args}$ ])) AS _(out)
14       UNION
15       :
16       UNION
17       SELECT  $n$  AS site, out FROM (replace(slice( $f, s_n$ ), [(g.out). $\overline{args}$ ])) AS _(out)
18     ),
19     calls(site, fanout, out, val) AS ( Calls
20       SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL:: $\tau$  AS val
21       FROM slices AS s
22       WHERE s.out  $\diamond \perp$ 
23     )
24     TABLE memoization
25     UNION ALL
26     SELECT *
27     FROM calls
28     WHERE NOT EXISTS (TABLE memoization)
29     UNION ALL
30     SELECT NULL::int AS site, 0 AS fanout, ROW((g.out). $\overline{args}$ ) AS out,
31           (replace(body( $f, [NULL::\tau, \dots, NULL::\tau]$ ),
32                    [(g.out). $\overline{args}$ ])) AS val
33     WHERE NOT EXISTS (TABLE memoization)
34     AND NOT EXISTS (TABLE calls) MemConstruct
35   ) AS edges(site, fanout, out, val)
36   WHERE g.fanout > 0 OR g.fanout IS NULL
37 )

```

Figure 5.25: Tweaking the template of Figure 4.8 in Chapter 4 enables memoization for compiled functional-style UDFs.

```

1 CREATE FUNCTION  $f(\overline{\text{args}})$  RETURNS  $\tau$ 
2 AS $$
3 WITH RECURSIVE call_graph(in,site,out,val) AS (
4   <see Figure 5.25> MemGraph
5 ),
6 base_cases(in,val,ref,site,fanout) AS (
7   SELECT g.in, g.val, g_ref.in, g_ref.site,g_ref.fanout
8   FROM call_graph AS g,
9        call_graph AS g_ref
10  WHERE g.fanout = 0
11        AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
12        AND g.in = g_ref.out RefBase
13 ),
14 evaluation(in,val,ref,site,fanout) AS (
15   <see Figure 5.4> RefEval
16 ),
17 store AS (
18   INSERT INTO memo
19   SELECT e.in, e.val
20   FROM evaluation AS e
21   ON CONFLICT DO NOTHING MemStore
22 )
23 SELECT e.val RefResult
24 FROM evaluation AS e
25 WHERE e.fanout IS NULL;
26 $$ LANGUAGE SQL VOLATILE STRICT;

```

Figure 5.26: The compiled SQL code with memoization enabled. As part of the evaluation process, store (intermediate) results of CTE evaluation in memo for later use in CTE call_graph.

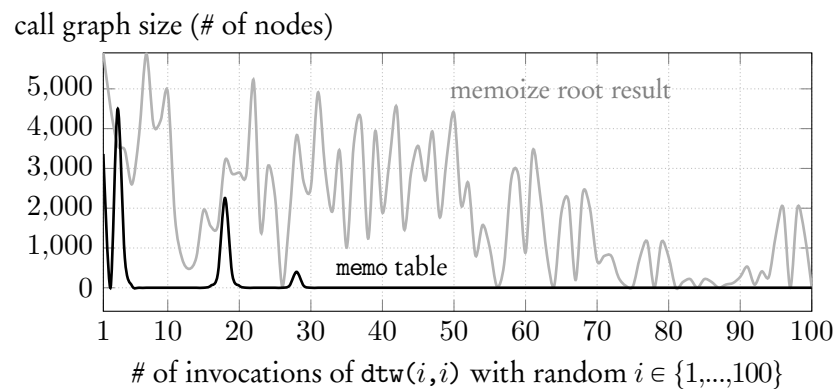


Figure 5.27: Series of $\text{dtw}(i,i)$ invocations: Re-using memo table entries effectively cuts down call graph size.

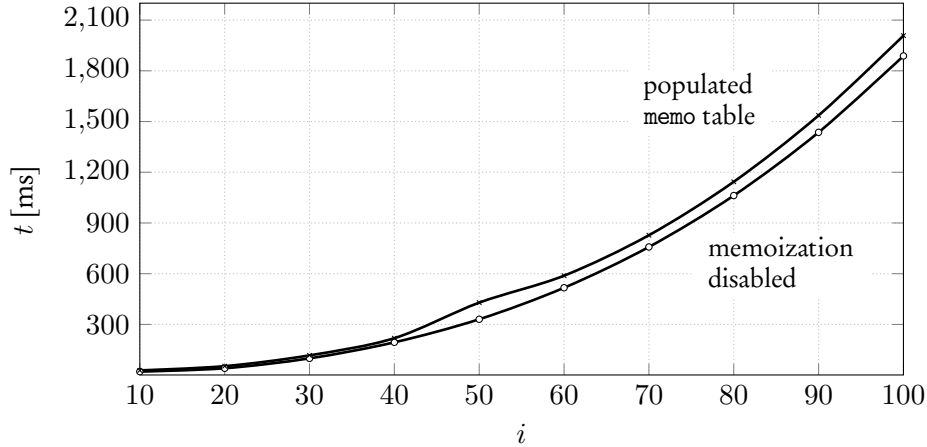


Figure 5.28: Compare evaluation of $\text{dtw}(i, i)$ with and without memoization enabled. Table `memo` is created with index on column `in` and has been previously populated with 10,000,000 rows unrelated to any calls $\text{dtw}(i, i)$.

5.5.2 INDEXING TABLE MEMO

Assume that memoization has been enabled over many subsequent function invocations and table `memo` collected quite the amount of rows — 10,000,000 in fact. Each node found in the process of call graph construction tries to look up a previously evaluated result in table `memo`. And every time this is done, this takes about 500 milliseconds without an index — a very steep price to pay. Thus, when enabling memoization we always create a B-tree index [59] on column `in` for table `memo` to ease the workload when memoization is enabled. With the index, even a quite sizable `memo` table which does not contain *any* rows that help shrink the call graph heavily mitigates the impact of a large, but useless, `memo` table. In fact, Figure 5.28 compares a function invocation of compiled $\text{dtw}(i, i)$ with increasing i and memoization enabled (—●—) to the same function invocation with memoization disabled (—○—). Here, table `memo` is populated with rows that never come up when constructing the call graph. Indeed, this index is always created in the following runtime experiments.

UDF	Recursion	Avg. Call Time [ms]			Call Times 100 invoc.s	Avg. Call Graph Size		memo
		root	memo table			root	memo table	
shortest	n-fold	443	12	(2.7%)		1,455	32	1,965
dtw	3-fold	327	23	(7.0%)		5,863	310	10,201
knapsack	2-fold	356	26	(7.3%)		7,857	412	20,301
needleman	3-fold	278	29	(10.4%)		3,979	251	8,291
paths	linear	138	9	(6.5%)		1,025	21	1,997
parcels	tv tail	207	15	(7.2%)		45,027	1,531	3,000
bom	tv n-fold	119	146	(122.7%)		6	1,067	131,055
split	2-fold	249	270	(108.4%)		5,022	5,022	334,748
vm	tail	264	358	(135.6%)		18,027	18,027	56

Table 5.1: A collection of SQL UDFs in functional style compiled with root memoization (root) and non-root memoization (memo table) enabled.

5.5.3 RUNTIME EXPERIMENTS

Figure 5.27 plots the call graph sizes we observed during a sequence of 100 invocations of $dtw(i, i)$ with random $i \in \{1, \dots, 100\}$. As expected, memoizing the top-most root call reduces the call graph sizes over time (\searrow). However, memoization of subgraph roots (\swarrow) is by more effective, bringing call graphs down to size 1 already after only a dozen calls. We repeated the random invocations sequence multiple times and reported average call graph sizes here.

The results in Table 5.1 report on runtime and call graph size averages over 100 random calls. We compare compiled use cases with memoization of *only* root calls to compiled use cases with memoization of all non-root results stored in the memo table. Each function is grouped into one of the following: recursion with correlated call sites (n-fold recursion, recall Section 5.1), recursion with fixed call sites (for example, 3-fold recursive function dtw), linear recursion and tail recursion. Prefix *tv* marks use cases that use table-valued return types.

- Compilation reduces 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 shortest, dtw,

knapsack, needleman, paths, and parcels can effectively reuse prior evaluation efforts while split and vm fail to do so. bom is table-valued and thus a special case we describe below.

- 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 100 invocations have been performed.

Taking a closer look at some of these use cases leads to interesting observations about the functions' behavior at runtime:

shortest: This is an optimal use case where memoization really shines. The non-root results stored in memo decrease the call graph size very early.

paths: A file system is structured as a directory tree. Thus, as the function climbs the directory tree to reconstruct the path, there is a high chance of overlap with previous calls improving runtimes for linear recursion.

parcels: Even though we only memoize root call results for tail-recursive functions, we still look for results of root calls already present in the memo table as we construct the call_graph. Indeed, especially this use case shows very promising results even though we store only root calls in memo. This is not generally the case for tail-recursive functions. however (as we will see with vm).

bom: For table-valued functions, enabling memoization may be detrimental. Whenever call graph construction finds table-valued results with many rows in memo, these rows are then *all* added to the call_graph CTE *increasing* the call graph size potentially by the number of rows. This increases the cost of the join with the call_graph CTE in each bottom-up traversal step. Indeed, we measure the impact this has with this use case (cf. the average call graph sizes for bom). Thus, the decision to enable memoization when dealing with table-valued functions is something the developer must carefully consider. It highly depends on the number of rows the developer expects for each table-valued result.

split: For this use case, none of the random calls have overlapping call graphs (unless it is the same call). Thus, we only pay for the overhead of memoization without its

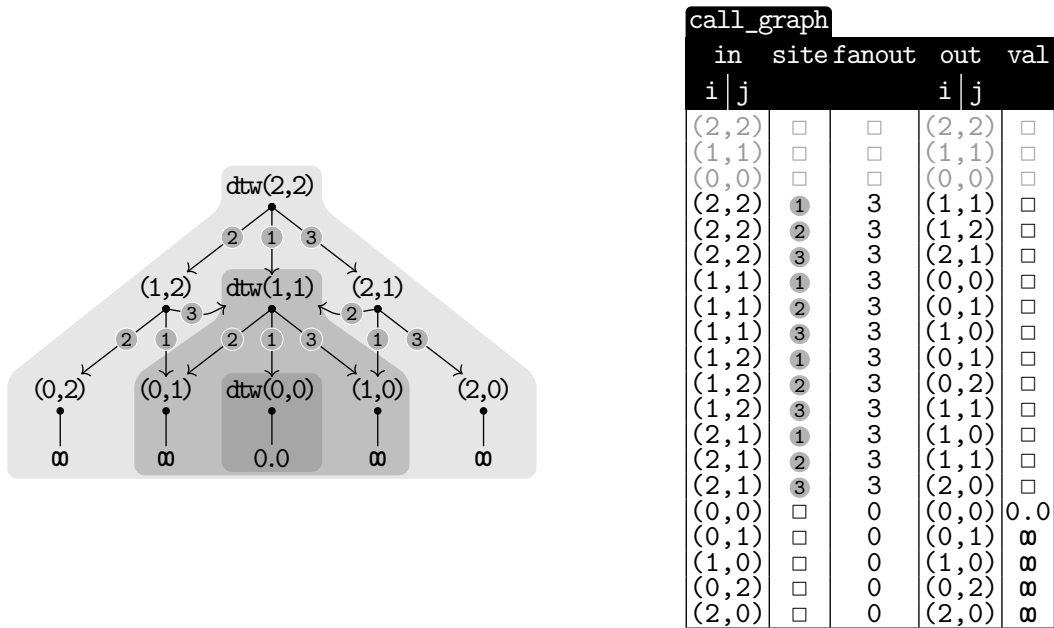
benefits (cf. the average call time of `split`). Some use cases like this and `vm` require the developer to consider the call graph structure a function produces before enabling memoization.

`vm`: As with `split`, each invocation always produces a unique call graph without overlap. Evaluation of `vm` either scans `memo` in each call graph construction step without success or exactly *once* at the very beginning of call graph construction.

5.5.4 SUMMARY

Memoization of non-root results store all intermediate results a function produces when traversing the call graph bottom-up. Indeed, enabling memoization for tail-recursive functions inherently only stores the root result when compiled. However, tail-recursive functions (when compiled) still may look up results in the `memo` table early in the process of call graph construction. Thus, they may still profit off of memoization in the long run, as we have seen with use case `parcels` in Table 5.1.

On the other hand: in some cases, the developer may want to hold back on enabling memoization. Functions where different calls rarely lead to any hits in the `memo` table during call graph constructions are left with the memoization overhead and without its benefits (see `split` and `vm`). Furthermore, table-valued functions that are not tail-recursive can also be a bad fit for memoization. The call graph size may increase overall when a huge table-valued result is found in the `memo` table, which makes the CTE scan of the `call_graph` CTE required in each step of the bottom-up traversal increasingly more expensive (see `bom`).



(a) Call graph.

(b) Tabular representation of the call graph.

Figure 5.29: Call graph of $\text{dtw}([(2,2), (1,1), (0,0)])$ and its tabular representation. The call graphs of $\text{dtw}(2,2)$, $\text{dtw}(1,1)$ and $\text{dtw}(0,0)$ collapse into one.

5.6 BATCHING

Batching is a technique derived from *data-parallel languages* [63]. Another technique, called *flattening*, also bears strong similarities in its concept of lifting functions to arbitrary orders [106]. However, batching is exclusively concerned with first-order flattening: $f : a \rightarrow b$ to $f' : [a] \rightarrow [b]$.

Assume that function f is invoked many times, say $f(x_1), \dots, f(x_n)$. Compiled without batching, each invocation constructs an independent call graph top to bottom. Compiling f with batching enabled changes the function from accepting one vector of arguments $f(x)$ at a time to accepting many vectors of arguments $f'([x, \dots, x])$. Thus, each individual function invocation $f(x_i)$ collapses into a single call $f'([x_1, \dots, x_n])$ that constructs a single call graph with up to n root nodes. Indeed, batching collapses all redundant subgraphs into one.

For example, take dtw (compiled with batching enabled) with a function invo-

		evaluation				
		in	val	ref	site	fanout
<i>i</i>		i j		i j		
1	(0,0)	0.0	(1,1)	1	3	
	(0,1)	∞	(1,1)	2	3	
	(1,0)	∞	(1,1)	3	3	
	(0,1)	∞	(1,2)	1	3	
	(0,2)	∞	(1,2)	2	3	
	(1,0)	∞	(2,1)	1	3	
	(2,0)	∞	(2,1)	3	3	
	(0,0)	0.0	(0,0)	□	□	
2	(0,1)	∞	(1,2)	1	3	
	(0,2)	∞	(1,2)	2	3	
	(1,0)	∞	(2,1)	1	3	
	(2,0)	∞	(2,1)	3	3	
	(1,1)	0.0	(2,2)	1	3	
	(1,1)	0.0	(1,2)	3	3	
	(1,1)	0.0	(2,1)	2	3	
	(1,1)	0.0	(1,1)	□	□	
3	(1,1)	0.0	(2,2)	1	3	
	(1,2)	1.0	(2,2)	2	3	
	(2,1)	0.0	(2,2)	3	3	
4	(2,2)	1.0	(2,2)	□	□	

(a) Bottom-up traversal.

		dtw	
		in	val
<i>i</i>	<i>j</i>		
(0,0)		0.0	
(1,1)		0.0	
(2,2)		1.0	

(b) Result table.

Figure 5.30: Bottom-up traversal and result table of batched call $\text{dtw}([(2,2), (1,1), (0,0)])$.

```

1 WITH RECURSIVE call_graph(in, site, fanout, out, val) AS (
2   SELECT args, NULL::int, NULL::bigint, args, NULL::τ
3   FROM unnest(f.[args]) AS args BatAnchor
4
5   UNION -- recursive UNION
6   SELECT g.out, edges.*
7   FROM call_graph AS g,
8   LATERAL (
9     WITH slices(site, out) AS (
10    SELECT 1 AS site, out FROM (replace(slice(f, s1), [(g.out).args])) AS _(out) Slices
11    UNION
12    :
13    SELECT n AS site, out FROM (replace(slice(f, sn), [(g.out).args])) AS _(out)
14   ),
15   calls(site, fanout, out, val) AS (
16    SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL::τ AS val
17    FROM slices AS s
18    WHERE s.out <> ⊥ Calls
19   )
20   TABLE calls
21   UNION ALL
22   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out,
23     (replace(body(f, [NULL::τ, ..., NULL::τ]),
24     [(g.out).args])) AS val
25   WHERE NOT EXISTS (TABLE calls) Construct
26 ) AS edges(site, fanout, out, val)
27 WHERE g.fanout > 0 OR g.fanout IS NULL Invoke
28 )

```

Figure 5.31: Tweaking the template of Figure 4.8 in Chapter 4 enables batching for compiled functional-style UDFs.

cation such as `dtw([(2,2), (1,1), (0,0)])`. The call graphs for calls `dtw(1,1)` and `dtw(0,0)` become subgraphs of `dtw(2,2)` (see Figure 5.29). This is not generally the case but highlights the potential benefits of batching.

5.6.1 TWEAKING THE SQL TEMPLATE

Except for the signature change of function f , compilation with batching enabled requires only very minor tweaks in two regions: **BatAnchor** and **BatResult**. We highlight these changes that lead to the templates in Figures 5.31 and 5.32:

BatAnchor The function `unnest([args])` [97] takes the array of arguments `[args]` and turns it into a table with one column where each row holds one of the call graph roots `args`.

BatResult The resulting query returns a table that matches the signature of f such that it returns a table where each row is a pair of root calls and results: `((in), (val))`.

Note that the batching tweaks can be applied independently of other (optional)

```

1 CREATE FUNCTION f( $\overline{\text{args}}$ ) RETURNS TABLE( $\overline{\text{args}}$ ,  $\tau$ )
2 AS $$
3 WITH RECURSIVE call_graph(in,site,fanout,out,val) AS (
4     <see Figure 5.31>
5     ),
6     base_cases(in,val,ref,site,fanout) AS (
7         SELECT g.in, g.val, g_ref.in, g_ref.site,g_ref.fanout
8         FROM call_graph AS g,
9              call_graph AS g_ref
10        WHERE g.fanout = 0
11              AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
12              AND g.in = g_ref.out
13        ),
14     evaluation(in,val,ref,site,fanout) AS (
15         <see Figure 5.4>
16     )
17     SELECT e.in, e.val
18     FROM evaluation AS e
19     WHERE e.fanout IS NULL;
20 $$ LANGUAGE SQL STABLE STRICT;

```

Figure 5.32: We insert Figure 5.31 as the `call_graph` CTE and apply minor tweaks to the function signature and `BatResult`.

compiler tweaks presented in this chapter. Templates with all tweaks enabled can be found in Appendix B, where their modular nature is highlighted.

5.6.2 RUNTIME EXPERIMENTS

The results in Table 5.2 report the total **call time** and total **call graph size** for a selection of compiled use cases with and without batching enabled. For the **arguments**, we use those found in each use case’s runtime experiment plots. For example, the input for UDF `dtw` is found in Figure 5.9. Many of the use cases profit from batching because of their overlapping call graphs over many calls: `shortest`, `dtw`, `knapsack`, `needleman`, and `paths`. We highlight a few use cases:

bom: High overlap between subgraphs allows the compiled function with batching enabled to discard 83% of the nodes. Indeed, as with other use cases, not all function calls construct highly overlapping call graphs. However, when they do, batching is a good way to profit from it in terms of runtime optimization.

vm: This (and `parcels`) are tail-recursive functions which both do not profit from batching. Each call graph is disjunct from one another, increasing the overall size of

UDF	Recursion	Arguments	Σ Call Time [ms]			Σ Call Graph Size	
			compiled	batched		compiled	batched
shortest	n-fold	Figure 5.12	5,002	906	(18.1%)	13,779	2,723
dtw	3-fold	Figure 5.9	6,400	1,909	(29.8%)	116,825	30,216
knapsack	2-fold	Figure 5.10	7,165	2,115	(29.5%)	156,680	40,417
needleman	3-fold	Figure 5.11	19,835	5,563	(28.0%)	115,114	29,915
paths	linear	Figure 5.19	27,847	6,756	(24.3%)	65,517	10,017
bom	tv n-fold	Figure 5.16	5,205	2,697	(51.8%)	1,003	171
parcels	tv tail	Figure 5.22	11,540	11,431	(99.0%)	2,779,480	2,779,480
vm	tail	Figure 5.21	3,493	3,232	(92.5%)	232,950	232,950
split	2-fold	Figure 5.7	1,744	4,762	(273.1%)	24,549	24,549

Table 5.2: A collection of SQL UDFs in functional style compiled with and without batching enabled.

the batched call graph. Indeed, because of the tail-recursive optimization, which removes the need for bottom-up traversal, we measure neither significant improvements nor declines in call times.

split: This function does not exhibit any partially overlapping subgraphs in each function invocation. Indeed, this means the batched function call and its bottom-up traversal must now deal with an overinflated call graph. Recall in Section 5.1.1, bottom-up traversal enforces a runtime complexity lower bound by the call graph size: $|\text{call_graph}|^2$. Without batching, each function call has (at most) a size of 12,287. Compare this to the single batched call graph with 24,549. As expected, the increased call graph size almost triples the call times of the batched function compared to the sum of the time each call needs. This represents a worst-case scenario for batching.

5.6.3 SUMMARY

Batching exploits functions whose call graphs over many independent invocations overlap, which can benefit execution times. However, this puts the burden of handling this change to the function arguments and return type (i.e., $f : a \rightarrow b$ to $f' : [a] \rightarrow [b]$) on the developer who maintains the surrounding queries which invoke this function.

Indeed, not all functions benefit from this optional optimization flag. A function,

such as `split`, exhibits no partially overlapping subgraphs. Batching significantly increases its call graph size, thus worsening the execution times.

Let us close this chapter with a few final remarks: The tweaks and improvements presented in this chapter give the developer many optimization switches that may improve runtime for their complex computation pushed inside the RDBMS. Indeed, with the exception of `ITERATE` (recall Section 5.4), the benefits of compiling functional-style UDFs are readily available in all RDBMS that implements SQL:1999 [101] which the compiler can target without intrusive changes to the RDBMS itself. However, in some cases, developers must understand the shape of their functional-style UDF's call graph structure before they decide to compile them. Functions, such as `split`, may be more suited to be implemented as a carefully hand-crafted CTE wherever performance concerns outweigh readability.

6

CONCLUSION

In this publication, we set out to measure the viability for developers to write functional-style UDFs, giving developers more ways to push their computation close to the data [98]. We found our answer on two fronts: readability from the viewpoint of the developer and runtime performance.

First, we measured the readability aspect of functional-style UDFs by performing a user study (Chapter 2). We found that functional-style UDFs are a welcome addition to the bag of tools of developers using SQL aside from recursive CTEs aiming to implement complex computations. Some developers may even unanimously prefer the use of functional-style for some computations. Indeed, when we let the participants decide whether to implement the algorithm to solve the 0-1 Knapsack problem [88] in functional-style or as a recursive CTE, not a single submission even tried implementing it using recursive CTEs. We suspect that other development tools, such as a debugger for functional-style UDFs, maybe one future branch of research to support developers in their endeavours.

Besides functional-style UDFs, other alternatives should also be studied further in this manner: Some of them are already published, where the developer is allowed to implement functions with complex control flows in PL/PgSQL [97] or Python, which

are then compiled to pure SQL functions [81, 76]. Another way to support recursive CTEs in their mission to allow developers to implement complex computations is to break up the rigid semantics of recursive CTEs. When evaluating recursive CTEs, each iteration step builds a new intermediate table from scratch, becoming the working table for the next iteration. And only if the intermediate table is empty, does evaluation stop. Current research asks if we could express some algorithms more elegantly if alternative semantics were available to the developer. One such alternative imagines that the working table is not built from scratch with each iteration. Instead, keep the working table at the start of the iteration and only insert new rows and update old ones in each iteration.

Second, we measured the runtime performance aspect of functional-style UDFs (Chapters 4 and 5). Running functional-style UDFs as is, comes with a list of restrictions for many RDBMS. SQLite3 does not support SQL UDFs in general [102]. MySQL 8.0's SQL UDFs do not allow for recursive self-involutions within their function bodies [90]. Microsoft SQL Server 2022 supports recursive self-involutions in SQL UDFs, but only with a hard-coded recursion depth limit of 32 [100]. Oracle 19c [95] and PostgreSQL 13 [97] allow for more meaningful usage of functional-style UDFs, where the recursion depth can go beyond the 10,000 mark. However, the recursion depth is still limited and its value *cannot* exceed the maximum stack size set by the operating system. On top of these restrictions, PostgreSQL does not optimize for recursion in functional-style UDFs as far as we can tell. Moreover, PostgreSQL repeatedly parses and plans the function body for each recursive call due to the absence of plan caching.

We chose to approach this problem by compiling functional-style UDFs which removes all recursive self-involutions (Chapter 4). Instead, the compiler targets recursive CTEs that ultimately evaluate the function in a two-phased approach that constructs the call graph and then evaluates it by traversing it bottom-up. If an RDBMS supports recursive CTEs, then this enables that system to support functional-style UDFs efficiently through SQL-to-SQL compilation. This applies to systems like SQLite3, which do not implement UDFs at all (compiler simply inlines the compiled body at the call site to obtain a function-free query), as well as MySQL, Microsoft SQL Server and Oracle, where compiled UDFs are not limited by the strin-

gent restrictions on recursion depth. In PostgreSQL, not only does compilation of functional-style UDFs work around recursion depth limits. It also allows many UDFs to evaluate significantly faster (compared to before compilation) without requiring intrusive changes to the underlying database engine. Indeed, this is due to the various optimizations this approach comes with discussed in Chapter 5.

In future work, we may enable the compiler to target other RDBMSs. Further optimization potential may still remain untapped, waiting to be discovered. One such work observes the benefits of different storage methods of the call graph at runtime. For example, Madeleine Mauz’s master thesis explored the benefits of storing the call graph into a hash table at runtime, allowing call graph traversal speedy lookup of parent nodes in the call graph. Thus, traversing the call graph more efficiently bottom-up.

In conclusion, as developers face larger growing data sets every day without an end in sight, so too must SQL provide more approachable options for developers looking to query computational results instead of *just* the data. Functional-style UDFs, as presented in this publication, mark another viable step towards this goal.



USER STUDY - ONLINE FORM

Introduction

Thank you for participating in this *anonymous user study!*

Like any programming language, SQL allows you to express complex computation in various (equivalent) forms. This survey studies two particular forms of authoring *user-defined SQL functions* (UDFs). Both forms are used to implement the same recursive programming problems—but the forms read differently:

- one uses recursive *CTEs* (`WITH RECURSIVE`),
- the other is based on *recursive* function invocation.

This survey compares these two forms in terms of *readability* and thus focuses on your ability to *read and understand* snippets of SQL code. **We therefore ask you to work through the study without the use of external programs: answer the questions below without executing the UDFs on a SQL database system.** Use pen and paper only (if required at all). Thank you!

The SQL dialect presented to you in this survey leans heavily on PostgreSQL 12. If any feature found in the following queries is new to you, feel free to look them up [here](#).

Important: We will also ask you for the time it took you to complete each task. As you complete each task, keep track of the required time and enter it in the boxes below. Please answer truthfully. Each task should take about 5 – 10 minutes, if you find yourself exceeding this timespan, please feel free to skip to the next task (in this case, please tick the corresponding box).

The entire survey should take about 30 – 45 minutes to complete. Once done, please make sure to press the *Submit your results!*-button at the end.

Before we begin, please list the **three (regular) programming languages** that you are most familiar with:

1.
2.
3.

I have had some exposure to Functional Programming:

Yes No

Part 1: Fibonacci Numbers

Graham et. al. define the *Fibonacci number function* in *Concrete Mathematics, 2nd edition* as:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \end{aligned} \quad (1)$$

We propose the following UDFs which aim to compute the *Fibonacci number* of n using SQL. You can assume that each of the UDFs are syntactically sound but only some of them implement the *Fibonacci number function* correctly.

1. Keep track of how long it takes for you to choose which of the following CTE-based UDFs implement the *Fibonacci number function* correctly (there may be more than one):

<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ WITH RECURSIVE fib(i, prev_n, curr_n) AS (VALUES (1, 0, 1) UNION SELECT f.i + 1, f.curr_n, f.curr_n - f.curr_n FROM fib AS f WHERE f.i <= n) SELECT f.curr_n FROM fib AS f WHERE f.i = n; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ WITH RECURSIVE fib(i, prev_n, curr_n) AS (SELECT 1, 0, 1 UNION SELECT f.i + 1, f.curr_n, f.prev_n + f.curr_n FROM fib AS f WHERE f.i BETWEEN 0 AND n) SELECT f.curr_n FROM fib AS f WHERE f.i < n; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ WITH RECURSIVE fib(i, prev_n, curr_n) AS (VALUES (1, 0, 1) UNION ALL SELECT f.i + 1, f.curr_n, f.prev_n + f.curr_n FROM fib AS f WHERE f.i <= n) SELECT f.prev_n FROM fib AS f ORDER BY f.i DESC LIMIT 1; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ WITH RECURSIVE fib(i, prev_n, curr_n) AS (VALUES (1, 0, 1) UNION ALL SELECT f.i + 1, f.curr_n, f.curr_n + f.curr_n FROM fib AS f WHERE f.i < n) SELECT f.curr_n FROM fib AS f ORDER BY f.i LIMIT 1; \$\$ LANGUAGE SQL;</pre>
--	---	---	---

Solving this task took about minutes.

Or, tick this () if you want to skip it.

2. Keep track of how long it takes for you to choose which of the following recursive UDFs implement the *Fibonacci number function* correctly (there may be more than one):

<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ SELECT 0 WHERE n <= 1 UNION ALL SELECT fib(n-1) + fib(n-2) WHERE n > 1; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ SELECT CASE WHEN n = 0 THEN 0 WHEN n = 1 THEN 1 ELSE fib(n-1) + fib(n-2) END; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ SELECT CASE WHEN n = 0 THEN 0 WHEN n = 1 THEN 1 ELSE fib(n-1) - fib(n-2) END; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION fib(n int) RETURNS int AS \$\$ SELECT 1 WHERE n <= 2 UNION ALL SELECT fib(n-1) + fib(n-2) WHERE n > 2; \$\$ LANGUAGE SQL;</pre>
---	---	---	---

Solving this task took about minutes.

Or, tick this () if you want to skip it.

Part 2: Greatest Common Divisor

Knuth et. al. define the calculation of the *greatest common divisor* in *The Art of Programming (vol. 2 Seminumerical Algorithms), 3rd edition* as:

$$\begin{aligned} gcd(n, 0) &= n \\ gcd(n, k) &= gcd(k, n \bmod k) \end{aligned} \quad (2)$$

We propose the following SQL UDFs which aim to compute the *greatest common divisor* of u and v . You can assume that each of the UDFs are syntactically sound but only some of them implement the *greatest common divisor function* correctly.

1. Keep track of how long it takes for you to choose which of the following recursive UDFs compute the *greatest common divisor* correctly (there may be more than one):

<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ SELECT CASE WHEN k = 0 THEN n ELSE gcd(k, MOD(n,k)) END; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ SELECT n WHERE k = 0 UNION ALL SELECT gcd(k, MOD(k,n)) WHERE k > 0; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ SELECT CASE WHEN n = 0 THEN k ELSE gcd(k, MOD(k,n)) END; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ SELECT k WHERE k = 0 UNION SELECT gcd(k, MOD(n,k)) WHERE k > 0; \$\$ LANGUAGE SQL;</pre>
---	---	---	---

Solving this task took about minutes.

Or, tick this () if you want to skip it.

2. Keep track of how long it takes for you to choose which of the following CTE-based UDFs compute the *greatest common divisor* correctly (there may be more than one):

<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ WITH RECURSIVE gcd(n,k) AS (SELECT n, k UNION ALL SELECT g.k, MOD(g.n,g.k) FROM gcd AS g WHERE g.k <> 0) SELECT g.n FROM gcd AS g WHERE g.k = 0; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ WITH RECURSIVE gcd(n,k) AS (SELECT n, k UNION SELECT g.k, MOD(g.n,g.k) FROM gcd AS g WHERE g.k > 0) SELECT g.n FROM gcd AS g ORDER BY g.k LIMIT 1; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ WITH RECURSIVE gcd(n,k) AS (SELECT n, k UNION ALL SELECT g.k, MOD(g.n,g.k) FROM gcd AS g WHERE g.k <> 0) SELECT g.k FROM gcd AS g WHERE g.k = 0; \$\$ LANGUAGE SQL;</pre>	<input type="checkbox"/> <pre>CREATE FUNCTION gcd(n int, k int) RETURNS int AS \$\$ WITH RECURSIVE gcd(n,k) AS (SELECT n, k UNION ALL SELECT g.k, MOD(g.n,g.k) FROM gcd AS g WHERE g.k = 0) SELECT g.n FROM gcd AS g ORDER BY g.k DESC LIMIT 1; \$\$ LANGUAGE SQL;</pre>
---	---	---	---

Solving this task took about minutes.

Or, tick this () if you want to skip it.

Part 3: Comprehension I

Consider the following table s:

```
CREATE TABLE s (  
  a serial PRIMARY KEY,  
  b float NOT NULL  
);
```

We define the following function $f(i, j, k)$ which evaluates over table s:

```
CREATE FUNCTION f(i int, j int, k float)  
RETURNS float AS $$  
SELECT CASE  
  WHEN i > j THEN k  
  ELSE (SELECT f(i+1, j, s.b + 0.5 * k)  
        FROM s  
        WHERE s.a = i)  
END;  
$$ LANGUAGE SQL;
```

Keep track on how long it takes you to describe (in 100 words or less) what the function $f(i \text{ int}, j \text{ int}, 0)$ does:

```
1 What does f(i int, j int, 0) do?  
2  
3 WRITE YOUR YOUR ANSWER HERE!
```

Solving this task took about minutes.
Or, tick this () if you want to skip it.

Part 4: Comprehension II

Consider the following table t:

```
CREATE TABLE t (  
  x int PRIMARY KEY,  
  y int REFERENCES t(x),  
  z int NOT NULL  
);
```

Consider the following function $g(a \text{ int})$ which evaluates over table t:

```
CREATE FUNCTION g(a int)  
RETURNS bigint AS $$  
WITH RECURSIVE  
  r(x, y, z) AS (  
    SELECT t.x, t.y, t.z  
    FROM t  
    WHERE t.x = a  
    UNION  
    SELECT r.x, t.y, t.z  
    FROM r, t  
    WHERE r.y = t.x  
  )  
SELECT SUM(r.z)  
FROM r;  
$$ LANGUAGE SQL;
```

Keep track on how long it takes you to describe (in 100 words or less) what the function $g(a \text{ int})$ does:

```
1 What does g(a int) do?  
2  
3 WRITE YOUR YOUR ANSWER HERE!
```

Solving this task took about minutes.
Or, tick this () if you want to skip it.

Part 5: Evaluation

Important: This part builds on functions found in **part 3** and **part 4** of this survey. Make sure to answer these parts first, before you continue here.

1. Consider the following instance for table s from part 3:

`INSERT INTO s(a,b) VALUES (1,4),(2,2.5),(3,1.5);`

Keep track on how long it takes you to compute (using pen and paper) the result of the function from part 3 when called as $f(1,3,0)$. Write down the result:

1 The result of $f(1,3,0)$?
2
3 WRITE YOUR YOUR ANSWER HERE!

Solving this task took about minutes.
Or, tick this () if you want to skip it.

2. Consider the following instance for table t from part 4:

`INSERT INTO t(x,y,z) VALUES (1,2,5),(2,4,3),(3,2,2),(4,3,1);`

Keep track on how long it takes you to compute (using pen and paper) the result of the function from part 4 when called as $g(4)$. Write down the result:

1 The result of $g(4)$?
2
3 WRITE YOUR YOUR ANSWER HERE!

Solving this task took about minutes.
Or, tick this () if you want to skip it.

Part 6: 0-1 Knapsack Problem

Martello et. al. define the 0-1 Knapsack Problem in *Knapsack Problems - Algorithms and Computer Implementations* as follows:

Consider items $i \in \{1, \dots, n\}$ where each item has a weight w_i and a value p_i . Then $knap(n, w)$ maximizes the sum of values of items that fit into a knapsack of weight w and is defined as:

$$knap(1, u) = 0$$
$$knap(k, u) = \begin{cases} knap(k-1, u) & , \text{if } w_k > u \\ \max(knap(k-1, u), knap(k-1, u - w_k) + p_k) & , \text{otherwise} \end{cases} \quad (3)$$

Keep track of how long it takes you to complete the SQL function $knap(n, w)$ which solves the 0-1 Knapsack problem either using a recursive or CTE-based UDF. Submit your solution in the following textbox in which we also provide to you the definition of table `items` which holds all items `i` and their respective weight `w` and value `p` to be used in this task.

```
1 CREATE TABLE items (  
2   i serial PRIMARY KEY,  
3   w int NOT NULL,  
4   p int NOT NULL  
5 );  
6  
7 CREATE FUNCTION knap(k int, u int)  
8 RETURNS int AS $$  
9 /* YOUR CODE HERE */  
10 $$ LANGUAGE SQL;
```

Solving this task took about minutes.

Or, tick this if you want to skip it.

All done?

Thank you for participating!

B

COMPLETE TEMPLATES

Templates used for compiling functional-style UDFs in general and with linear and tail recursion optimizations (see Sections 5.1, 5.3 and 5.4). Each has all features (table-valued return type, memoization, and batching) enabled can be found in Figures B.1 to B.3. In each Figure, we highlight the independent code regions that enable table-valued return types (■), memoization (▨), and batching (▩). These features are discussed in detail in Chapter 5.

```

1 CREATE FUNCTION f(args) RETURNS TABLE(args, vals)
2 AS $$
3 WITH RECURSIVE call_graph(in,site,fanout,out,vals, "empty?",rid) AS (
4   SELECT args, NULL::int, NULL::bigint, args, NULL, false, NULL::bigint
5   FROM unnest(f.args) AS args
6   UNION -- recursive UNION
7   SELECT g.out, edges.*
8   FROM call_graph AS g,
9   LATERAL (
10    WITH memoization(site,fanout,out,vals, "empty?",rid) AS (
11     SELECT NULL::int, 0, m.in, m.vals, m."empty?", m.rid
12     FROM memo AS m
13     WHERE g.out = m.in
14    ),
15    slices(site,out) AS (
16     SELECT 1 AS site, out FROM (replace(slice(f, s1), [(g.out).args])) AS _(out)
17     UNION
18     ⋮
19     UNION
20     SELECT n AS site, out FROM (replace(slice(f, sn), [(g.out).args])) AS _(out)
21    ),
22    calls(site,fanout,out,vals, "empty?",rid) AS (
23     SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL, false, NULL::bigint
24     FROM slices AS s
25     WHERE s.out <> 1
26    ),
27    values(vals) AS (
28     replace(body(f, [(VALUES NULL), ..., (VALUES NULL)]), [(g.out).args])
29    )
30   TABLE memoization
31   UNION ALL
32   SELECT *
33   FROM calls
34   WHERE NOT EXISTS (TABLE memoization)
35   UNION ALL
36   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out,
37     v.vals, false, ROW_NUMBER() OVER ()
38   FROM values AS v
39   WHERE NOT EXISTS (TABLE memoization)
40   AND NOT EXISTS (TABLE calls)
41   UNION ALL
42   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out, NULL, true, 1
43   WHERE NOT EXISTS (TABLE memoization)
44   AND NOT EXISTS (TABLE calls)
45   AND NOT EXISTS (TABLE values)
46   ) AS edges(site,fanout,out,vals, "empty?",rid)
47 WHERE g.fanout > 0 OR g.fanout IS NULL
48 ),

```

⟨Continued on next page⟩

⟨Continued from previous page⟩

```
49 base_cases(in,vals,"empty?",rid,ref,site,fanout) AS (  
50   SELECT g.in, g.vals, g."empty?", g.rid, g_ref.in, g_ref.site, g_ref.fanout  
51   FROM   call_graph AS g,  
52         call_graph AS g_ref  
53   WHERE g.fanout = 0  
54   AND   (g_ref.fanout > 0 OR g_ref.fanout IS NULL)  
55   AND   g.in = g_ref.out  
56 ),  
57 evaluation(in,vals,"empty?",rid,ref,site,fanout) AS (  
58   TABLE base_cases  
59   UNION ALL -- recursive UNION ALL  
60   (WITH e(in,vals,"empty?",rid,ref,site,fanout) AS (TABLE evaluation),  
61     returns(in,vals,"empty?",rid) AS (  
62       SELECT go.in, result.*  
63       FROM (SELECT e.ref  
64             FROM e  
65             WHERE e.rid = 1  
66             GROUP BY e.ref, e.fanout  
67             HAVING COUNT(*) = e.fanout  
68             ) AS go(in),  
69       LATERAL (  
70         WITH result(vals) AS (  
71           replace(body(f, [lookupTbl(f, 1), ..., lookupTbl(f, n)]),  
72             [(go.in).args])  
73         )  
74         SELECT r.*, false, ROW_NUMBER() OVER () FROM result AS r  
75         UNION ALL  
76         SELECT NULL, true, 1 WHERE NOT EXISTS (TABLE result)  
77         ) AS result(vals,"empty?",rid)  
78     )  
79   SELECT *  
80   FROM e  
81   WHERE e.fanout IS NOT NULL  
82   AND   NOT EXISTS (SELECT FROM returns AS r WHERE r.in = e.ref)  
83   UNION ALL  
84   SELECT r.*, g.in, g.site, g.fanout  
85   FROM returns AS r, call_graph AS g  
86   WHERE r.in = g.out  
87 )),  
88 store AS (  
89   INSERT INTO memo  
90   SELECT e.in, e.vals, e."empty?", e.rid  
91   FROM evaluation AS e  
92   ON CONFLICT DO NOTHING  
93 )  
94 SELECT e.in, e.vals  
95 FROM evaluation AS e  
96 WHERE e.fanout IS NULL  
97 AND NOT e."empty?";  
98 $$ LANGUAGE SQL VOLATILE STRICT;
```

Figure B.1: SQL code emitted when compiling functional-style UDFs with table-valued return type (■), memoization (▨) and batching (▧) enabled.

```

1 CREATE FUNCTION f(args) RETURNS TABLE(args, vals)
2 AS $$
3 WITH RECURSIVE call_graph(in,site,fanout,out,vals, "empty?",rid) AS (
4   SELECT args, NULL::int, NULL::bigint, args, NULL, false, NULL::bigint
5   FROM unnest(f.args) AS args
6   UNION ALL -- recursive UNION ALL
7   SELECT g.out, edges.*
8   FROM call_graph AS g,
9   LATERAL (
10    WITH memoization(site,fanout,out,vals, "empty?",rid) AS (
11     SELECT NULL::int, 0, m.in, m.vals, m."empty?", m.rid
12     FROM memo AS m
13     WHERE g.out = m.in
14    ),
15    slices(site,out) AS (
16     SELECT 1 AS site, out FROM (replace(slice(f, s1), [(g.out).args])) AS _(out)
17     UNION
18     ⋮
19     UNION
20     SELECT n AS site, out FROM (replace(slice(f, sn), [(g.out).args])) AS _(out)
21    ),
22    calls(site,fanout,out,vals, "empty?",rid) AS (
23     SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL, false, NULL::bigint
24     FROM slices AS s
25     WHERE s.out <> 1
26    ),
27    values(vals) AS (
28     replace(body(f, [(VALUES NULL), ..., (VALUES NULL)]), [(g.out).args])
29    )
30   TABLE memoization
31   UNION ALL
32   SELECT *
33   FROM calls
34   WHERE NOT EXISTS (TABLE memoization)
35   UNION ALL
36   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out,
37     v.vals, false, ROW_NUMBER() OVER ()
38   FROM values AS v
39   WHERE NOT EXISTS (TABLE memoization)
40   AND NOT EXISTS (TABLE calls)
41   UNION ALL
42   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out, NULL, true, 1
43   WHERE NOT EXISTS (TABLE memoization)
44   AND NOT EXISTS (TABLE calls)
45   AND NOT EXISTS (TABLE values)
46   ) AS edges(site,fanout,out,vals, "empty?",rid)
47 WHERE g.fanout > 0 OR g.fanout IS NULL
48 ),

```

⟨Continued on next page⟩

⟨Continued from previous page⟩

```
49 base_cases(in,vals,"empty?",rid,ref,site,fanout) AS (  
50   SELECT g.in, g.vals, g."empty?", g.rid, g_ref.in, g_ref.site, g_ref.fanout  
51   FROM   call_graph AS g,  
52         call_graph AS g_ref  
53   WHERE  g.fanout = 0  
54   AND    (g_ref.fanout > 0 OR g_ref.fanout IS NULL)  
55   AND    g.in = g_ref.out  
56 ),  
57 evaluation(in,vals,"empty?",rid,ref,site,fanout) AS (  
58   TABLE base_cases  
59   UNION ALL -- recursive UNION ALL  
60   (WITH e(in,vals,"empty?",rid,ref,site,fanout) AS (TABLE evaluation),  
61     returns(in,vals,"empty?",rid) AS (  
62       SELECT go.in, result.*  
63       FROM (SELECT e.ref  
64             FROM e  
65             WHERE e.rid = 1  
66             GROUP BY e.ref, e.fanout  
67             HAVING COUNT(*) = e.fanout  
68             ) AS go(in),  
69       LATERAL (  
70         WITH result(vals) AS (  
71           replace(body(f, [lookupTbl(f, 1), ..., lookupTbl(f, n)]),  
72             [(go.in).args])  
73         )  
74         SELECT r.*, false, ROW_NUMBER() OVER () FROM result AS r  
75         UNION ALL  
76         SELECT NULL, true, 1 WHERE NOT EXISTS (TABLE result)  
77         ) AS result(vals,"empty?",rid)  
78     )  
79   SELECT r.*, g.in, g.site, g.fanout  
80   FROM   returns AS r, call_graph AS g  
81   WHERE  r.in = g.out  
82 )),  
83 store AS (  
84   INSERT INTO memo  
85   SELECT e.in, e.vals, e."empty?", e.rid  
86   FROM   evaluation AS e  
87   ON CONFLICT DO NOTHING  
88 )  
89 SELECT e.in, e.vals  
90 FROM   evaluation AS e  
91 WHERE  e.fanout IS NULL  
92 AND    NOT e."empty?";  
93 $$ LANGUAGE SQL VOLATILE STRICT;
```

Figure B.2: SQL code emitted when compiling linear recursive functional-style UDFs with table-valued return type (■), memoization (▨) and batching (▧) enabled.

```

1 CREATE FUNCTION f(args) RETURNS TABLE(args, vals)
2 AS $$
3 WITH RECURSIVE call_graph(in,site,fanout,out,vals, "empty?",rid) AS (
4   SELECT args, NULL::int, NULL::bigint, args, NULL, false, NULL::bigint
5   FROM unnest(f.args) AS args
6   UNION ALL -- recursive UNION ALL
7   SELECT g.out, edges.*
8   FROM call_graph AS g,
9   LATERAL (
10    WITH memoization(site,fanout,out,vals, "empty?",rid) AS (
11     SELECT NULL::int, 0, m.in, m.vals, m."empty?", m.rid
12     FROM memo AS m
13     WHERE g.out = m.in
14    ),
15    slices(site,out) AS (
16     SELECT 1 AS site, out FROM (replace(slice(f, s1), [(g.out).args])) AS _(out)
17     UNION
18     ⋮
19     UNION
20     SELECT n AS site, out FROM (replace(slice(f, sn), [(g.out).args])) AS _(out)
21    ),
22    calls(site,fanout,out,vals, "empty?",rid) AS (
23     SELECT s.site, COUNT(*) OVER () AS fanout, s.out, NULL, false, NULL::bigint
24     FROM slices AS s
25     WHERE s.out <> 1
26    ),
27    values(vals) AS (
28     replace(body(f, [(VALUES NULL), ..., (VALUES NULL)]), [(g.out).args])
29    )
30   TABLE memoization
31   UNION ALL
32   SELECT *
33   FROM calls
34   WHERE NOT EXISTS (TABLE memoization)
35   UNION ALL
36   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out,
37     v.vals, false, ROW_NUMBER() OVER ()
38   FROM values AS v
39   WHERE NOT EXISTS (TABLE memoization)
40   AND NOT EXISTS (TABLE calls)
41   UNION ALL
42   SELECT NULL::int AS site, 0 AS fanout, ROW((g.out).args) AS out, NULL, true, 1
43   WHERE NOT EXISTS (TABLE memoization)
44   AND NOT EXISTS (TABLE calls)
45   AND NOT EXISTS (TABLE values)
46   ) AS edges(site,fanout,out,vals, "empty?",rid)
47 WHERE g.fanout > 0 OR g.fanout IS NULL
48 ),

```

⟨Continued on next page⟩

⟨Continued from previous page⟩

```
49 store AS (  
50     INSERT INTO memo  
51     SELECT g.in, g.vals, g."empty?", g.rid  
52     FROM   call_graph AS g  
53     WHERE  g.fanout = 0  
54     ON CONFLICT DO NOTHING  
55 )  
56 SELECT g.in, g.vals  
57 FROM   call_graph AS g  
58 WHERE  g.fanout = 0  
59 AND    NOT g."empty?";  
60 $$ LANGUAGE SQL VOLATILE STRICT;
```

Figure B.3: SQL code emitted when compiling tail recursive functional-style UDFs with table-valued return type (■), memoization (▨) and batching (▧) enabled.

C

SELECTED FUNCTIONAL-STYLE SQL UDFs

We hand-picked a selection of functional-style SQL UDFs and list their SQL implementation (Appendix C), their compiled SQL function (Section C.2), and their hand-crafted recursive CTE formulation (Section C.3).

C.1 IMPLEMENTATION

Figures C.1 to C.4 show the functional-style SQL UDFs for their respective selected use-cases. Each figure also shows the required table and type definitions.


```

1 CREATE TABLE X (
2   t int PRIMARY KEY,
3   x real
4 );

5 CREATE TABLE Y (
6   t int PRIMARY KEY,
7   y real
8 );

9 CREATE FUNCTION dtw(i int, j int) RETURNS real
10 AS $$
11   CASE
12     WHEN i=0 AND j=0 THEN 0.0
13     WHEN i=0 OR j=0 THEN ∞ -- 'Infinity'::real
14     ELSE (SELECT abs(Z.x - Z.y)
15           +
16           LEAST(1dtw(i-1, j-1),
17                2dtw(i-1, j ),
18                3dtw(i , j-1))
19           FROM (X JOIN Y
20                ON ((X.t,Y.t) = (i,j))) AS Z)
21   END;
22 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.1: The functional-style dtw. ¹, ², and ³ mark the recursive call sites. Function dtw is 3-fold recursive.

```

1 CREATE FUNCTION split(x real, y real) RETURNS int
2 AS $$
3   CASE
4     WHEN ABS(x-y) <= 1.0 THEN 1
5     ELSE 1split(x,x+ABS(x-y)/2.0)
6           +
7           2split(x+ABS(x-y)/2.0,y)
8   END;
9 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.2: The functional-style split. ¹ and ² mark the recursive call sites. Function split is 2-fold recursive.

```

1 CREATE TABLE materials (
2   part    int,
3   sub     int,
4   quantity int
5 );

6 CREATE FUNCTION bom(part int)
7 RETURNS TABLE(part int, sub int, quantity int)
8 AS $$
9   SELECT m.part, m.sub, m.quantity
10  FROM materials AS m
11  WHERE m.part = part
12  UNION ALL
13  SELECT b.part, b.sub, m.quantity * b.quantity
14  FROM materials AS m JOIN LATERAL ❶ bom(m.sub) AS b
15  ON (m.part = part);
16 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.3: The functional-style bom. ❶ marks the recursive call site. Function bom is n-fold recursive with a table-valued return type.

```

1 -- Currently supported VM instruction set
2 CREATE TYPE opcode AS ENUM (
3   'lod', -- lod t, x      load literal x into target register Rt
4   'mov', -- mov t, s      move from source register Rs to target register Rt
5   'jeq', -- jeq t, s, @a  if Rt = Rs, jump to location a, else fall through
6   'jmp', -- jmp @a        jump to location a
7   'add', -- add t, s1, s2  Rt <- Rs1 + Rs2
8   'mul', -- mul t, s1, s2  Rt <- Rs1 * Rs2
9   'div', -- div t, s1, s2  Rt <- Rs1 / Rs2
10  'mod', -- mod t, s1, s2  Rt <- Rs1 mod Rs2
11  'hlt'  -- hlt s          halt program, result is register Rs
12 );

13 -- VM instructions
14 CREATE TYPE instruction AS (
15   loc int, -- location
16   opc opcode, -- opcode
17   reg1 int, -- \
18   reg2 int, -- } up to three work registers
19   reg3 int  -- /
20 );
21 CREATE TABLE program OF instruction;

22 CREATE FUNCTION vm(ins instruction, regs int[])
23 RETURNS int AS $$
24   CASE ins.opc
25     WHEN 'lod' THEN ①vm((SELECT p
26                          FROM program AS p
27                          WHERE p.loc = ins.loc+1),
28                          regs[:ins.reg1-1] || ins.reg2 || regs[ins.reg1+1:])
29     WHEN 'mov' THEN ②vm((SELECT p
30                          FROM program AS p
31                          WHERE p.loc = ins.loc+1),
32                          regs[:ins.reg1-1] || regs[ins.reg2] || regs[ins.reg1+1:])
33     WHEN 'jeq' THEN ③vm((SELECT p
34                          FROM program AS p
35                          WHERE p.loc = CASE WHEN regs[ins.reg1] = regs[ins.reg2]
36                                             THEN ins.reg3
37                                             ELSE ins.loc + 1
38                                             END),
39                          regs)
40     WHEN 'jmp' THEN ④vm((SELECT p
41                          FROM program AS p
42                          WHERE p.loc = ins.reg1),
43                          regs)
44     WHEN 'add' THEN ⑤vm((SELECT p
45                          FROM program AS p
46                          WHERE p.loc = ins.loc+1),
47                          regs[:ins.reg1-1] || regs[ins.reg2] + regs[ins.reg3] || regs[ins.reg1+1:])
48     WHEN 'mul' THEN ⑥vm((SELECT p
49                          FROM program AS p
50                          WHERE p.loc = ins.loc+1),
51                          regs[:ins.reg1-1] || regs[ins.reg2] * regs[ins.reg3] || regs[ins.reg1+1:])
52     WHEN 'div' THEN ⑦vm((SELECT p
53                          FROM program AS p
54                          WHERE p.loc = ins.loc+1),
55                          regs[:ins.reg1-1] || regs[ins.reg2] / regs[ins.reg3] || regs[ins.reg1+1:])
56     WHEN 'mod' THEN ⑧vm((SELECT p
57                          FROM program AS p
58                          WHERE p.loc = ins.loc+1),
59                          regs[:ins.reg1-1] || regs[ins.reg2] % regs[ins.reg3] || regs[ins.reg1+1:])
60     WHEN 'hlt' THEN regs[ins.reg1]
61   END;
62 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.4: The functional-style `vm`. ①, ②, ..., ⑧ mark the recursive call sites. Function `vm` is tail-recursive. Each CASE-branch directly calls exactly one call site.

C.2 COMPILATION

Compiling the functional-style UDF replaces their function body as described in Chapters 4 and 5. The functions in Figures C.6 to C.8 present the compiled functional-style UDFs of Figures C.1 to C.4 respectively. In each figure, we marked the sections which were modified by *slice*, *replace*, and *body* (■) and how it was modified (■).

```

1 CREATE FUNCTION dtw(i int, j int) RETURNS real
2 AS $$
3 WITH RECURSIVE
4 call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS (
5   SELECT i, j, NULL :: int, NULL :: bigint, i, j, NULL::real
6     UNION -- Recursive UNION
7     SELECT g.out_i, g.out_j, edges.*
8     FROM call_graph AS g,
9     LATERAL (
10      WITH
11      slices(site, out) AS (
12 SELECT ① AS site, out FROM (
13 SELECT *
14 FROM (SELECT (NULL, false) :: lifted_args) AS _
15 WHERE (g.out_i = 0 AND g.out_j = 0)
16 UNION ALL
17 SELECT *
18 FROM (SELECT (NULL, false) :: lifted_args) AS _
19 WHERE NOT (g.out_i = 0 AND g.out_j = 0) AND (g.out_i = 0 OR g.out_j = 0)
20 UNION ALL
21 SELECT *
22 FROM (SELECT (ROW(g.out_i - 1, g.out_j - 1), true) :: lifted_args
23 FROM (X JOIN Y
24 ON ((X.t,Y.t) = (g.out_i,g.out_j))) AS Z) AS _
25 WHERE NOT ((g.out_i = 0 AND g.out_j = 0) OR (g.out_i = 0 OR g.out_j = 0))
26 ) AS _(out) UNION SELECT ② AS site, out FROM (
27 SELECT *
28 FROM (SELECT (NULL, false) :: lifted_args) AS _
29 WHERE (g.out_i = 0 AND g.out_j = 0)
30 UNION ALL
31 SELECT *
32 FROM (SELECT (NULL, false) :: lifted_args) AS _
33 WHERE NOT (g.out_i = 0 AND g.out_j = 0) AND (g.out_i = 0 OR g.out_j = 0)
34 UNION ALL
35 SELECT *
36 FROM (SELECT (ROW(g.out_i - 1, g.out_j), true) :: lifted_args
37 FROM (X JOIN Y
38 ON ((X.t,Y.t) = (g.out_i,g.out_j))) AS Z) AS _
39 WHERE NOT ((g.out_i = 0 AND g.out_j = 0) OR (g.out_i = 0 OR g.out_j = 0))
40 ) AS _(out) UNION SELECT ③ AS site, out FROM (
41 SELECT *
42 FROM (SELECT (NULL, false) :: lifted_args) AS _
43 WHERE (g.out_i = 0 AND g.out_j = 0)
44 UNION ALL
45 SELECT *
46 FROM (SELECT (NULL, false) :: lifted_args) AS _
47 WHERE NOT (g.out_i = 0 AND g.out_j = 0) AND (g.out_i = 0 OR g.out_j = 0)
48 UNION ALL
49 SELECT *
50 FROM (SELECT (ROW(g.out_i, g.out_j - 1), true) :: lifted_args
51 FROM (X JOIN Y
52 ON ((X.t,Y.t) = (g.out_i,g.out_j))) AS Z) AS _
53 WHERE NOT ((g.out_i = 0 AND g.out_j = 0) OR (g.out_i = 0 OR g.out_j = 0))
54 ) AS _(out)),
55 calls(site, fanout, i, j, val) AS (
56   SELECT s.site, COUNT(*) OVER (), (s.out).args.i, (s.out).args.j, NULL::real
57   FROM slices AS s
58   WHERE (s.out).not_bottom)
59 TABLE calls
60 UNION ALL
61 SELECT NULL :: int, 0, g.out_i, g.out_j, (
62 SELECT CASE
63   WHEN g.out_i = 0 AND g.out_j = 0 THEN 0::real
64   WHEN g.out_i = 0 OR g.out_j = 0 THEN 'infinity':real
65   ELSE (SELECT abs(Z.x - Z.y) + LEAST(NULL::real,NULL::real,NULL::real)
66 FROM (X JOIN Y
67 ON ((X.t,Y.t) = (g.out_i,g.out_j))) AS Z)
68 END)
69 WHERE NOT EXISTS (TABLE calls)
70 ) AS edges(site, fanout, i, j, val)
71 WHERE g.fanout > 0 OR g.fanout IS NULL),

```

⟨Continued on next page⟩

⟨Continued from previous page⟩

```
72 base_cases(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (  
73   SELECT g.in_i, g.in_j, g.val, g.ref.in_i, g.ref.in_j, g.ref.site, g.ref.fanout  
74   FROM   call_graph AS g, call_graph AS g_ref  
75   WHERE  g.fanout = 0 AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)  
76   AND    (g.in_i, g.in_j) = (g_ref.out_i, g_ref.out_j)  
77 ),  
78 evaluation(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (  
79   TABLE base_cases  
80   UNION ALL (  
81     WITH e AS (TABLE evaluation),  
82     returns(in_i, in_j, val) AS (  
83       SELECT go.in_i, go.in_j, (  
84         SELECT CASE replace(body(dtw,[lookup(dtw,⓪),lookup(dtw,⓪),lookup(dtw,⓪)]),[go.in_i,go.in_j])  
85         WHEN go.in_i = 0 AND go.in_j = 0 THEN 0::real  
86         WHEN go.in_i = 0 OR go.in_j = 0 THEN 'infinity'::real  
87         ELSE (SELECT abs(Z.x - Z.y) + LEAST(  
88           (SELECT e.val  
89             FROM e  
90             WHERE (e.in_i, e.in_j) = (go.in_i - 1, go.in_j - 1)  
91                 AND (e.ref_i, e.ref_j) = (go.in_i, go.in_j)  
92                 AND e.ref_site = ①),  
93           (SELECT e.val  
94             FROM e  
95             WHERE (e.in_i, e.in_j) = (go.in_i - 1, go.in_j )  
96                 AND (e.ref_i, e.ref_j) = (go.in_i, go.in_j)  
97                 AND e.ref_site = ②),  
98           (SELECT e.val  
99             FROM e  
100            WHERE (e.in_i, e.in_j) = (go.in_i , go.in_j - 1)  
101                AND (e.ref_i, e.ref_j) = (go.in_i, go.in_j)  
102                AND e.ref_site = ③))  
103            FROM X JOIN Y  
104            ON ((X.t,Y.t) = (go.in_i,go.in_j))) AS Z)  
105         END)  
106       FROM (  
107         SELECT e.ref_i, e.ref_j  
108         FROM e  
109         GROUP BY (e.ref_i, e.ref_j), e.ref_fanout  
110         HAVING COUNT(*) = e.ref_fanout  
111       ) AS go(in_i, in_j)  
112     )  
113     SELECT *  
114     FROM e  
115     WHERE NOT e.ref_fanout IS NULL  
116     AND NOT EXISTS (SELECT  
117                       FROM returns AS r  
118                       WHERE (r.in_i, r.in_j) = (e.ref_i, e.ref_j))  
119     UNION ALL  
120     SELECT r.in_i, r.in_j, r.val, g.in_i, g.in_j, g.site, g.fanout  
121     FROM returns AS r, call_graph AS g  
122     WHERE (r.in_i, r.in_j) = (g.out_i, g.out_j)  
123   ))  
124   SELECT e.val  
125   FROM evaluation AS e  
126   WHERE e.ref_fanout IS NULL;  
127 $$ LANGUAGE SQL STABLE STRICT;
```

Figure C.5: The compiled function dtw using the template described in Section 5.1.

```

1 CREATE FUNCTION split(x real, y real) RETURNS int
2 AS $$
3 WITH RECURSIVE
4 call_graph(in_x, in_y, site, fanout, out_x, out_y, val) AS (
5   SELECT x, y, NULL :: int, NULL :: bigint, x, y, NULL :: int
6   UNION -- Recursive UNION
7   SELECT g.out_x, g.out_y, edges.*
8   FROM call_graph AS g,
9   LATERAL (
10    WITH
11    slices(site, out) AS (
12     SELECT ① AS site, out FROM (
13     SELECT *
14     FROM (SELECT (SELECT (NULL, false) :: lifted_args)) AS _
15     WHERE ABS(g.out_x-g.out_y) <= 1.0
16     UNION ALL
17     SELECT *
18     FROM (SELECT (ROW(g.out_x,g.out_x+ABS(g.out_x-g.out_y)/2.0), true) :: lifted_args) AS _
19     WHERE NOT ABS(g.out_x-g.out_y) <= 1.0
20     ) AS _(out) UNION SELECT ② AS site, out FROM (
21     SELECT *
22     FROM (SELECT (SELECT (NULL, false) :: lifted_args)) AS _
23     WHERE ABS(g.out_x-g.out_y) <= 1.0
24     UNION ALL
25     SELECT *
26     FROM (SELECT (ROW(g.out_x+ABS(g.out_x-g.out_y)/2.0,g.out_y), true) :: lifted_args) AS _
27     WHERE NOT ABS(g.out_x-g.out_y) <= 1.0
28     ) AS _(out)),
29   calls(site, fanout, x, y, val) AS (
30     SELECT s.site, COUNT(*) OVER (), (s.out).args.x, (s.out).args.y, NULL :: int
31     FROM slices AS s
32     WHERE (s.out).not_bottom
33   )
34   TABLE calls
35   UNION ALL
36   SELECT NULL :: int, 0, g.out_x, g.out_y, (
37   SELECT CASE
38     WHEN ABS(g.out_x-g.out_y) <= 1.0 THEN 1
39     ELSE NULL :: real
40     +
41     NULL :: real
42   END)
43   WHERE NOT EXISTS (TABLE calls)
44   ) AS edges(site, fanout, x, y, val)
45   WHERE g.fanout > 0 OR g.fanout IS NULL),

```

<Continued on next page>

⟨Continued from previous page⟩

```
46 base_cases(in_x, in_y, val, ref_x, ref_y, ref_site, ref_fanout) AS (  
47   SELECT g.in_x, g.in_y, g.val, g.ref.in_x, g.ref.in_y, g.ref.site, g.ref.fanout  
48   FROM   call_graph AS g, call_graph AS g_ref  
49   WHERE  g.fanout = 0 AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)  
50   AND    (g.in_x, g.in_y) = (g_ref.out_x, g_ref.out_y)  
51 ),  
52 evaluation(in_x, in_y, val, ref_x, ref_y, ref_site, ref_fanout) AS (  
53   TABLE base_cases  
54   UNION ALL (  
55     WITH e AS (TABLE evaluation),  
56     returns(in_x, in_y, val) AS (  
57       SELECT go.in_x, go.in_y, (  
58         SELECT CASE replace(body(slice,[lookup(slice,0),lookup(slice,0)]),[go.in_x,go.in_y])  
59         WHEN ABS(go.in_x-go.in_y) <= 1.0 THEN 1  
60         ELSE (SELECT e.val  
61              FROM   e  
62              WHERE  (e.in_x , e.in_y) = (go.in_x,go.in_x+ABS(go.in_x-go.in_y)/2.0)  
63              AND    (e.ref_x, e.ref_y) = (go.in_x, go.in_y)  
64              AND    e.ref_site = 1)  
65              +  
66              (SELECT e.val  
67              FROM   e  
68              WHERE  (e.in_x , e.in_y) = (go.in_x+ABS(go.in_x-go.in_y)/2.0,go.in_y)  
69              AND    (e.ref_x, e.ref_y) = (go.in_x, go.in_y)  
70              AND    e.ref_site = 2)  
71         END)  
72       FROM (  
73         SELECT e.ref_x, e.ref_y  
74         FROM   e  
75         GROUP BY (e.ref_x, e.ref_y), e.ref_fanout  
76         HAVING COUNT(*) = e.ref_fanout  
77       ) AS go(in_x, in_y)  
78     )  
79     SELECT *  
80     FROM   e  
81     WHERE NOT e.ref_fanout IS NULL  
82     AND   NOT EXISTS (SELECT  
83                   FROM   returns AS r  
84                   WHERE  (r.in_x, r.in_y) = (e.ref_x, e.ref_y))  
85     UNION ALL  
86     SELECT r.in_x, r.in_y, r.val, g.in_x, g.in_y, g.site, g.fanout  
87     FROM   returns AS r, call_graph AS g  
88     WHERE  (r.in_x, r.in_y) = (g.out_x, g.out_y)  
89   ))  
90   SELECT e.val  
91   FROM   evaluation AS e  
92   WHERE  e.ref_fanout IS NULL;  
93   $$ LANGUAGE SQL STABLE STRICT;
```

Figure C.6: The compiled function split using the template described in Section 5.1.


```

1 CREATE FUNCTION bom(part int)
2 RETURNS TABLE(part int, sub int, quantity int)
3 AS $$
4 WITH RECURSIVE
5 call_graph(in_part, site, fanout, out_part, part, sub, quantity, "empty?", rid) AS (
6   SELECT part, NULL :: int, NULL :: bigint, part, NULL::int, NULL::int, NULL::int, false, NULL :: bigint
7     UNION -- Recursive UNION
8     SELECT g.out_part, edges.*
9     FROM call_graph AS g,
10      LATERAL (
11        WITH
12        slices(site, out) AS (
13          SELECT 1 AS site, out FROM (
14            SELECT b.*
15            FROM materials AS m JOIN LATERAL (SELECT (ROW(m.sub), true) :: lifted_args) AS b
16              ON m.part = g.out_part
17          ) AS _(out),
18          calls(site, fanout, out_part, part, sub, quantity, "empty?", rid) AS (
19            SELECT s.site, COUNT(*) OVER (), (s.out).args.part, NULL::int, NULL::int,
20              NULL::int, false, NULL :: bigint
21            FROM slices AS s
22            WHERE (s.out).not_bottom),
23          values(part, sub, quantity) AS (
24            SELECT m.part, m.sub, m.quantity
25            FROM materials AS m
26            WHERE m.part = g.out_part
27            UNION ALL
28            SELECT b.part, b.sub, m.quantity * b.quantity
29            FROM materials AS m JOIN LATERAL (VALUES (NULL::int,NULL::int,NULL::int)) AS b(part,sub,quantity)
30              ON m.part = g.out_part)
31      TABLE calls
32      UNION ALL
33      SELECT NULL :: int, 0, g.out_part,
34         v.*, false, ROW_NUMBER() OVER ()
35      FROM values AS v
36      WHERE NOT EXISTS (TABLE calls)
37      UNION ALL
38      SELECT NULL :: int, 0, g.out_part, NULL::int, NULL::int, NULL::int, true, 1
39      WHERE NOT EXISTS (TABLE calls)
40      AND NOT EXISTS (TABLE values)
41 ) AS edges(site, fanout, out_part, part, sub, quantity, "empty?", rid),

```

⟨Continued on next page⟩

⟨Continued from previous page⟩

```
42 base_cases(in_part, part, sub, quantity, "empty?", rid, ref_part, ref_site, ref_fanout) AS (  
43   SELECT g.in_part, g.part, g.sub, g.quantity, g."empty?", g.rid, g.ref.in_part, g.ref.site, g.ref.fanout  
44   FROM   call_graph AS g, call_graph AS g_ref  
45   WHERE  g.fanout = 0 AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)  
46   AND    (g.in_part) = (g_ref.out_part)  
47 ),  
48 evaluation(in_part, part, sub, quantity, "empty?", rid, ref_part, ref_site, ref_fanout) AS (  
49   TABLE base_cases  
50   UNION ALL (  
51     WITH e AS (TABLE evaluation),  
52     returns(in_part, part, sub, quantity, "empty?", rid) AS (  
53       SELECT go.in_part, result.*  
54       FROM (  
55         SELECT e.ref_part  
56         FROM   e  
57         WHERE  e.rid = 1  
58         GROUP BY (e.ref_part), e.ref_fanout  
59         HAVING COUNT(*) = e.ref_fanout  
60       ) AS go(in_part),  
61       LATERAL (  
62         WITH result(part, sub, quantity) AS (  
63           SELECT m.part, m.sub, m.quantity replace(body(bom,[lookup(bom,⓪)]),[go.in_part])  
64           FROM   materials AS m  
65           WHERE  m.part = go.in_part  
66           UNION ALL  
67           SELECT b.part, b.sub, m.quantity * b.quantity  
68           FROM   materials AS m JOIN LATERAL (SELECT e.part, e.sub, e.quantity  
69             FROM   e  
70             WHERE  NOT e."empty?"  
71             AND    (e.in_part) = (m.sub)  
72             AND    (e.ref_part) = (go.in_part)  
73             AND    e.ref_site = 1) AS b(part,sub,quantity)  
74             ON m.part = go.in_part  
75         )  
76         SELECT r.*, false, ROW_NUMBER() OVER ()  
77         FROM   result AS r  
78         UNION ALL  
79         SELECT NULL::int, NULL::int, NULL::int, true, 1  
80         WHERE  NOT EXISTS (TABLE result)  
81       ) AS result(part, sub, quantity, "empty?")  
82     )  
83     SELECT *  
84     FROM   e  
85     WHERE  NOT e.ref_fanout IS NULL  
86     AND    NOT EXISTS (SELECT  
87       FROM   returns AS r  
88       WHERE  (r.in_part) = (e.ref_part))  
89     UNION ALL  
90     SELECT r.in_part, r.part, r.sub, r.quantity, r."empty?", r.rid, g.in_part, g.site, g.fanout  
91     FROM   returns AS r, call_graph AS g  
92     WHERE  (r.in_part) = (g.out_part)  
93 ))  
94 SELECT e.part, e.sub, e.quantity  
95 FROM   evaluation AS e  
96 WHERE  e.ref_fanout IS NULL  
97 AND    NOT e."empty?";  
98 $$ LANGUAGE SQL STABLE STRICT;
```

Figure C.7: The compiled function `bom` with table-valued return type using the template described in Section 5.2.

```

1 CREATE FUNCTION vm(ins instruction, regs int[])
2 RETURNS int AS $$
3 WITH RECURSIVE
4 call_graph(in_ins, in_regs, site, fanout, out_ins, out_regs, val) AS (
5   SELECT ins, regs, NULL :: int, NULL :: bigint, ins, regs, NULL :: int
6     UNION ALL
7     SELECT g.in_ins, g.in_regs, edges.*
8     FROM call_graph AS g,
9     LATERAL (
10      WITH
11      slices(site, out) AS (
12 SELECT 1 AS site, out FROM (
13 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).loc+1),
14   (g.out_regs)[: (g.out_ins).reg1-1] || (g.out_ins).reg2 || (g.out_regs)[(g.out_ins).reg1+1:]),
15   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'lod'
16   UNION ALL
17   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
18 ) AS _(out) UNION SELECT 2 AS site, out FROM (
19 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).loc+1),
20   (g.out_regs)[: (g.out_ins).reg1-1] || (g.out_ins).reg2 || (g.out_regs)[(g.out_ins).reg1+1:]),
21   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'mov'
22   UNION ALL
23   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
24 ) AS _(out) UNION SELECT 3 AS site, out FROM (
25 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p
26   WHERE p.loc = CASE WHEN (g.out_regs)[(g.out_ins).reg1] = (g.out_regs)[(g.out_ins).reg2]
27     THEN (g.out_ins).reg3 ELSE (g.out_ins).loc + 1 END),
28   (g.out_regs)), true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'jeq'
29   UNION ALL
30   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
31 ) AS _(out) UNION SELECT 4 AS site, out FROM (
32 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).reg1), (g.out_regs)),
33   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'jmp'
34   UNION ALL
35   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
36 ) AS _(out) UNION SELECT 5 AS site, out FROM (
37 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).loc+1),
38   (g.out_regs)[: (g.out_ins).reg1-1] || (g.out_regs)[(g.out_ins).reg2] +
39   (g.out_ins).reg3 || (g.out_regs)[(g.out_ins).reg1+1:]),
40   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'add'
41   UNION ALL
42   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
43 ) AS _(out) UNION SELECT 6 AS site, out FROM (
44 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).loc+1),
45   (g.out_ins).reg1-1] || (g.out_ins).reg2] *
46   (g.out_ins).reg3 || (g.out_ins).reg1+1:]),
47   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'mul'
48   UNION ALL
49   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
50 ) AS _(out) UNION SELECT 7 AS site, out FROM (
51 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).loc+1),
52   (g.out_ins).reg1-1] || (g.out_ins).reg2] /
53   (g.out_ins).reg3 || (g.out_ins).reg1+1:]),
54   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'div'
55   UNION ALL
56   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
57 ) AS _(out) UNION SELECT 8 AS site, out FROM (
58 (SELECT * FROM (SELECT (ROW((SELECT p FROM program AS p WHERE p.loc = (g.out_ins).loc+1),
59   (g.out_ins).reg1-1] || (g.out_ins).reg2] %
60   (g.out_ins).reg3 || (g.out_ins).reg1+1:]),
61   true) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'mod'
62   UNION ALL
63   SELECT * FROM (SELECT (NULL, false) :: lifted_args) AS _ WHERE (g.out_ins).opc = 'hlt')
64 ) AS _(out)
65 ),

```

<Continued on next page>

⟨Continued from previous page⟩

```
50 calls(site, fanout, ins, regs, val) AS (  
51   SELECT s.site, 1, (s.out).args.ins, (s.out).args.regs, NULL :: int  
52   FROM   slices AS s  
53   WHERE  (s.out).not_bottom  
54 )  
55 TABLE calls  
56 UNION ALL  
57 SELECT NULL :: int, 0, g.out_ins, g.out_regs, (  
58 SELECT CASE (g.out_ins).opc  
59   WHEN 'lod' THEN NULL :: int  
60   WHEN 'mov' THEN NULL :: int  
61   WHEN 'jeq' THEN NULL :: int  
62   WHEN 'jmp' THEN NULL :: int  
63   WHEN 'add' THEN NULL :: int  
64   WHEN 'mul' THEN NULL :: int  
65   WHEN 'div' THEN NULL :: int  
66   WHEN 'mod' THEN NULL :: int  
67   WHEN 'hlt' THEN (g.out_regs)[(g.out_ins).reg1]  
68 END  
69 )  
70   WHERE NOT EXISTS (TABLE calls)  
71 ) AS edges(site, fanout, ins, regs, val)  
72 WHERE g.fanout > 0 OR g.fanout IS NULL  
73 )  
74 SELECT g.val  
75 FROM   call_graph AS g  
76 WHERE  g.fanout = 0;  
77 $$ LANGUAGE SQL STABLE STRICT;
```

Figure C.8: The compiled tail-recursive function `vm` using the template described in Section 5.4.

C.3 HAND-CRAFTED RECURSIVE CTEs

Figures C.9 to C.12 present the hand-crafted recursive CTEs formulation of each use-case found in Figures C.1 to C.4 without recursive self-involutions. These hand-crafted translations have been meticulously optimized and are used for runtime performance comparison throughout this publication.

```

1 CREATE TABLE X (
2   t int PRIMARY KEY,
3   x real
4 );

5 CREATE TABLE Y (
6   t int PRIMARY KEY,
7   y real
8 );

9 CREATE FUNCTION dtw(i int, j int) RETURNS real
10 AS $$
11 WITH RECURSIVE
12 warp(i,j,val) AS (
13   (SELECT X.t, Y.t, abs(X.x - Y.y)
14    FROM X, Y
15    ORDER BY X.t, Y.t
16    LIMIT 1)
17 UNION
18   SELECT step.i, step.j, MIN(step.val)
19   FROM (
20     SELECT step.i, step.j, warp.val + step.val
21     FROM warp, (VALUES (1,1),(0,1),(1,0)) AS d(i,j),
22     LATERAL (
23       SELECT warp.i+d.i, warp.j+d.j, abs(X.x - Y.y)
24       FROM X, Y
25       WHERE (X.t,Y.t) = (warp.i+d.i,warp.j+d.j)
26     ) AS step(i,j,val)
27     WHERE step.i <= dtw.i AND step.j <= dtw.j
28   ) AS step(i,j,val)
29   -- GROUP BY is a carefully placed optimization.
30   GROUP BY step.i, step.j)
31 SELECT MIN(warp.val) AS dtw
32 FROM warp
33 WHERE (warp.i,warp.j) = (dtw.i,dtw.j);
34 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.9: The hand-crafted SQL UDF dtw. Written by Denis Hirn.

```

1 CREATE FUNCTION split(x real, y real) RETURNS int
2 AS $$
3 WITH RECURSIVE
4 sp(x,y) AS (
5   SELECT split.x, split.y
6   UNION ALL
7   SELECT a,b
8   FROM   sp AS s,
9   LATERAL (SELECT s.x, s.x+ABS(s.x-s.y)/2.0
10          UNION ALL
11          SELECT s.x+ABS(s.x-s.y)/2.0,s.y) AS _(a,b)
12  WHERE  ABS(s.x-s.y) > 1.0
13 )
14 SELECT COUNT(*)
15 FROM   sp AS s
16 WHERE  ABS(s.x-s.y) <= 1.0;
17 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.10: The hand-crafted SQL UDF split.

```

1 CREATE TABLE materials (
2   part    int,
3   sub     int,
4   quantity int
5 );
6 CREATE FUNCTION bom(part int)
7 RETURNS TABLE(part int, sub int, quantity int)
8 AS $$
9 WITH RECURSIVE
10 included(top, part, sub, quantity) AS (
11   SELECT bom.part, m.part, m.sub, m.quantity
12   FROM   materials AS m
13   WHERE  bom.part = m.part
14   UNION ALL
15   SELECT i.top, m.part, m.sub, i.quantity * m.quantity
16   FROM   included AS i, materials AS m
17   WHERE  i.sub = m.part
18 )
19 SELECT i.top, i.sub, i.quantity :: int
20 FROM   included AS i, parts AS p
21 WHERE  i.part = p.id;
22 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.11: The hand-crafted SQL UDF bom without recursive calls. This function is loosely based on the sample function found in the PostgreSQL documentation [97].

```

1 -- Currently supported VM instruction set
2 CREATE TYPE opcode AS ENUM (
3   'lod', -- lod t, x      load literal x into target register Rt
4   'mov', -- mov t, s      move from source register Rs to target register Rt
5   'jeq', -- jeq t, s, @a  if Rt = Rs, jump to location a, else fall through
6   'jmp', -- jmp @a        jump to location a
7   'add', -- add t, s1, s2 Rt <- Rs1 + Rs2
8   'mul', -- mul t, s1, s2 Rt <- Rs1 * Rs2
9   'div', -- div t, s1, s2 Rt <- Rs1 / Rs2
10  'mod', -- mod t, s1, s2 Rt <- Rs1 mod Rs2
11  'hlt', -- hlt s          halt program, result is register Rs
12 );

13 -- VM instructions
14 CREATE TYPE instruction AS (
15   loc int, -- location
16   opc opcode, -- opcode
17   reg1 int, -- \
18   reg2 int, -- } up to three work registers
19   reg3 int -- /
20 );
21 CREATE TABLE program OF instruction;

22 CREATE FUNCTION vm(ins instruction, regs int[])
23 RETURNS int
24 AS $$
25 WITH RECURSIVE
26 run(ins, regs) AS (
27   SELECT vm.ins, vm.reg3
28   UNION ALL
29   SELECT p, n.reg3
30 FROM   run AS r, program AS p,
31 LATERAL (
32   SELECT r.reg3[: (r.ins).reg1-1] || (r.ins).reg2 || r.reg3[(r.ins).reg1+1:] WHERE (r.ins).opc = 'lod'
33   UNION ALL
34   SELECT r.reg3[: (r.ins).reg1-1] || r.reg3[(r.ins).reg2] || r.reg3[(r.ins).reg1+1:] WHERE (r.ins).opc = 'mov'
35   UNION ALL
36   SELECT r.reg3 WHERE (r.ins).opc = 'jeq'
37   UNION ALL
38   SELECT r.reg3 WHERE (r.ins).opc = 'jmp'
39   UNION ALL
40   SELECT r.reg3[: (r.ins).reg1-1] || r.reg3[(r.ins).reg2]+r.reg3[(r.ins).reg3] || r.reg3[(r.ins).reg1+1:]
41   WHERE (r.ins).opc = 'add'
42   UNION ALL
43   SELECT r.reg3[: (r.ins).reg1-1] || r.reg3[(r.ins).reg2]*r.reg3[(r.ins).reg3] || r.reg3[(r.ins).reg1+1:]
44   WHERE (r.ins).opc = 'mul'
45   UNION ALL
46   SELECT r.reg3[: (r.ins).reg1-1] || r.reg3[(r.ins).reg2]/r.reg3[(r.ins).reg3] || r.reg3[(r.ins).reg1+1:]
47   WHERE (r.ins).opc = 'div'
48   UNION ALL
49   SELECT r.reg3[: (r.ins).reg1-1] || r.reg3[(r.ins).reg2]%r.reg3[(r.ins).reg3] || r.reg3[(r.ins).reg1+1:]
50   WHERE (r.ins).opc = 'mod'
51 ) AS n(reg3)
52 WHERE (r.ins).opc <> 'hlt'
53 AND   p.loc = CASE (r.ins).opc
54         WHEN 'jeq' THEN CASE WHEN r.reg3[(r.ins).reg1] = r.reg3[(r.ins).reg2]
55                             THEN (r.ins).reg3
56                             ELSE (r.ins).loc+1 END
57         WHEN 'jmp' THEN (r.ins).reg1
58         WHEN 'hlt' THEN (r.ins).loc
59         ELSE (r.ins).loc+1
60     END)
61 SELECT r.reg3[(r.ins).reg1] FROM run AS r WHERE (r.ins).opc = 'hlt';
62 $$ LANGUAGE SQL STABLE STRICT;

```

Figure C.12: The hand-crafted SQL UDF vm.

D

ADDITIONAL USE CASES

D.1 BINOMIAL COEFFICIENT

Function `binomial(n, k)` implements the recursive formula to compute the binomial coefficient defined as:

$$\begin{aligned} \text{binomial}(n, 0) &= 1 \\ \text{binomial}(n, n) &= 1 \\ \text{binomial}(n, k) &= \text{binomial}(n - 1, k) + \text{binomial}(n - 1, k - 1) \end{aligned} \quad . \quad (\text{binomial})$$

Function `binomial` is 2-fold recursive. Figure D.1 compares execution times of `binomial` before and after compilation.

D.2 FLOYD-WARSHALL ALGORITHM

Given a graph with n nodes, where $w_{i \rightarrow j}$ denotes the weight of an edge $i \rightarrow j$ between two nodes v_i and v_j with $i, j \in 1, \dots, n$. Function `floyd(n, i, j)` finds the length of the shortest path from node v_i to v_j and is recursively defined based on Cormen et al. [67] as:

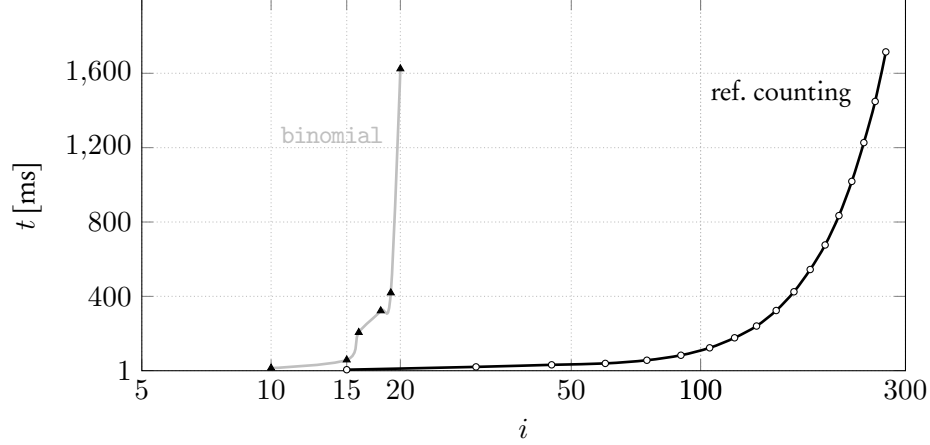


Figure D.1: Evaluating $\text{binomial}(i, \lfloor \frac{i}{4} \rfloor)$.

$$\begin{aligned} \text{floyd}(i, j, 0) &= w_{i \rightarrow j} \\ \text{floyd}(i, j, k) &= \min \left\{ \begin{array}{l} \text{floyd}(k-1, i, j) \\ \text{floyd}(k-1, i, k) + \text{floyd}(k-1, k, j) \end{array} \right\}. \end{aligned} \quad (\text{floyd})$$

Function `floyd` is 3-fold recursive. Figure D.2 compares execution times of `floyd` before and after compilation.

D.3 LONGEST COMMON SUBSEQUENCE

Function `lcs(1,1)` finds the length of the longest common subsequence of string s and t where $|s| = |t| = n$. We denote $s[k]$ to access a single character in s at position $k \in 1, \dots, n$. For example: For $s = \text{acd}$ and $t = \text{abc}$, $\text{lcs}_{s,t}(1,1) = 2$ because ac is the longest common subsequence. The function is recursively defined as:

$$\begin{aligned} \text{lcs}(i, n) &= 0 \\ \text{lcs}(n, j) &= 0 \\ \text{lcs}(i, j) &= \begin{cases} 1 + \text{lcs}(i+1, j+1) & , \text{if } s[i] = t[j] . \\ \min \left\{ \begin{array}{l} \text{lcs}(i+1, j) \\ \text{lcs}(i, j+1) \end{array} \right\} & , \text{otherwise} \end{cases} \end{aligned} \quad (\text{lcs})$$

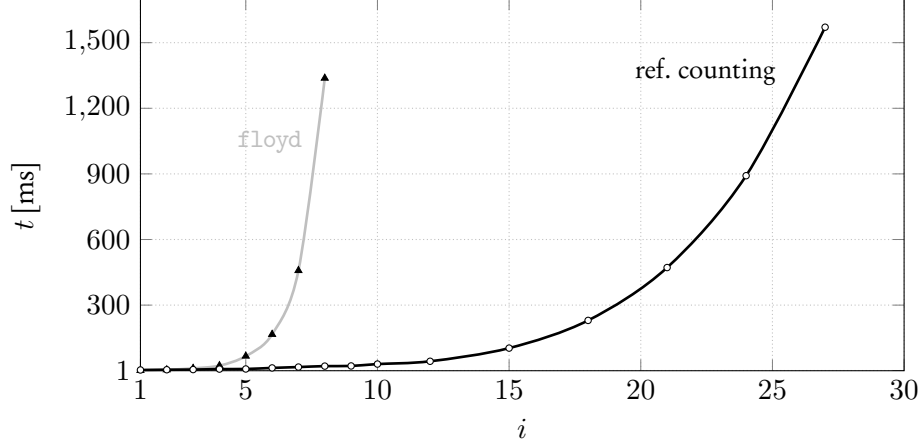


Figure D.2: Evaluating `floyd(i,i,1)`.

Function `lcs` is 2-fold recursive. Figure D.3 compares execution times of `lcs` before and after compilation.

D.4 REACHABILITY

Function `reach(vs, vt)` returns `TRUE`, if `vt` can be reached starting from `vs` in a DAG with `n` nodes, where `s, t ∈ 1, …, n`. Edge `es→t` connects `vs` with `vt`. Function `reach` is `n`-fold and recursively defined as:

$$\begin{aligned} \text{reach}(v_i, v_i) &= \text{TRUE} \\ \text{reach}(v_i, v_j) &= \bigvee_{e_{i \rightarrow k}} \text{reach}(v_k, v_j) \end{aligned} \quad (\text{reach})$$

Its call site is correlated and thus function `reach` is `n`-fold recursive. The DAG is set up so each node has a fanout of 1 or 2 and an average fanout of ~ 1.5 . To reach `vn` from `v1` `reach` passes through all nodes `v2, v3, …, vn-1` **at least once**. Figure D.4 compares execution times of `reach` before and after compilation.

D.5 FINITE STATE MACHINE

Function `fsm(p, s, n)` returns `TRUE`, if input `p` with length `n` is accepted by a finite state machine starting at state `s`. The transition function `q(p, s, n)` returns the next state

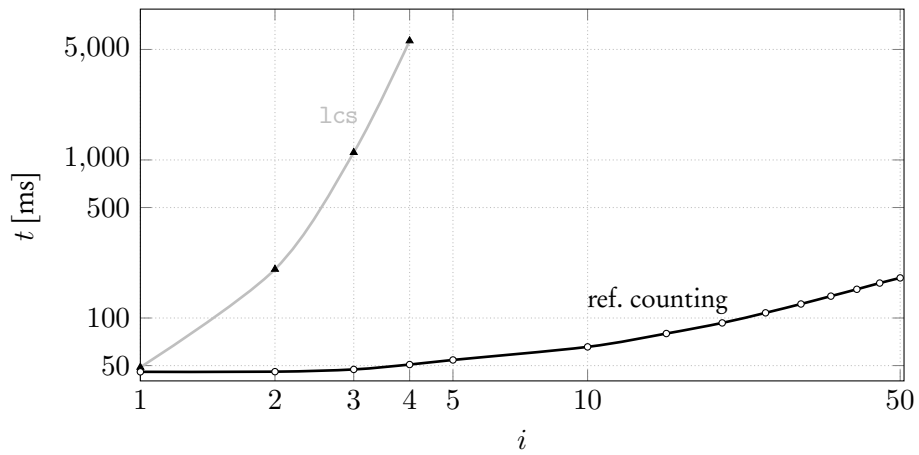


Figure D.3: Evaluating $\text{lcs}(s_1, s_2)$. Both strings s_1 and s_2 of length 100 contain different characters at i places.

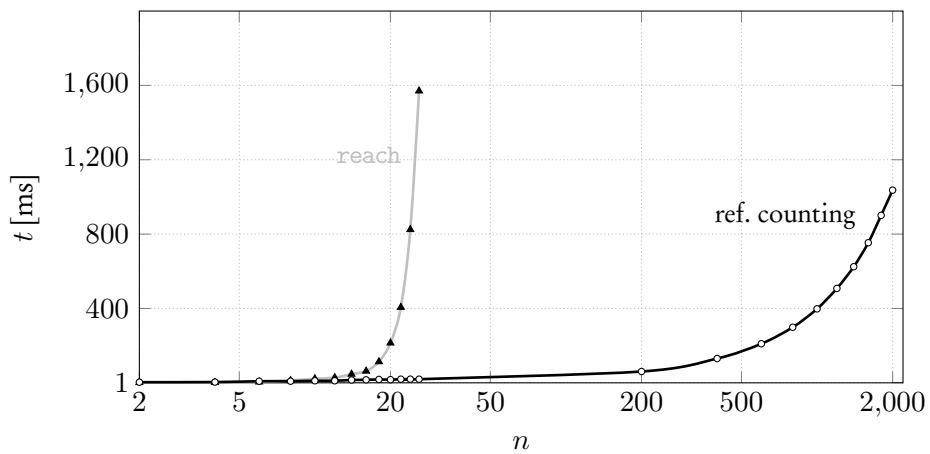


Figure D.4: Evaluating $\text{reach}(v_1, v_n)$.

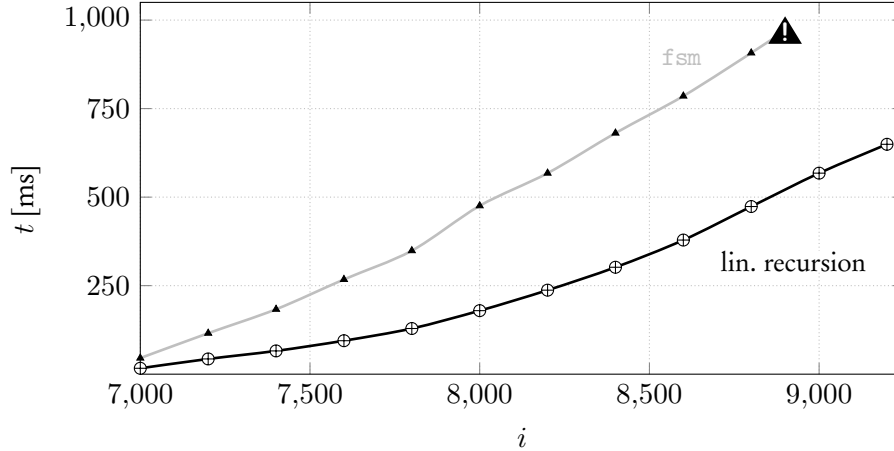


Figure D.5: Evaluating `fsm(p)` returns true, if protein sequence p of length i is accepted by a finite state machine. With increasing i , evaluation of `fsm` before compilation evaluation terminates prematurely due to stack overflow (▲).

following state s with input p at position n . Function `is_final(s)` only returns TRUE, if state s is an accepting state. Function `default(e, d)` returns the result of expression e , unless e evaluates to \emptyset in which case it returns default value d . Function `fsm` is recursively defined as:

$$\begin{aligned} \text{fsm}(p, s, 0) &= \text{is_final}(s) \\ \text{fsm}(p, s, n) &= \text{default}(\{\text{fsm}(p, s', n - 1) \mid s' = q(p, s, n)\}, \text{FALSE}) \end{aligned} \quad (\text{fsm})$$

Function `fsm` is linear recursive. Figure D.5 compares execution times of `fsm` before and after compilation.

D.6 BOUNDING BOX

Function `bbox(p_c, p_g, b)` produces the *minimum bounding box* of a 2D object. The *minimum bounding box algorithm* slightly modifies the marching cubes algorithm [86] to compute the bounding box of a 2D object. Function `move(p)` takes a 2D point on the edge of a 2D object and moves it along the edge counterclockwise. Function `box(p_1, p_2)` returns a 2D box from two 2D points p_1 and p_2 . Binary operator $b_1 \boxplus b_2$

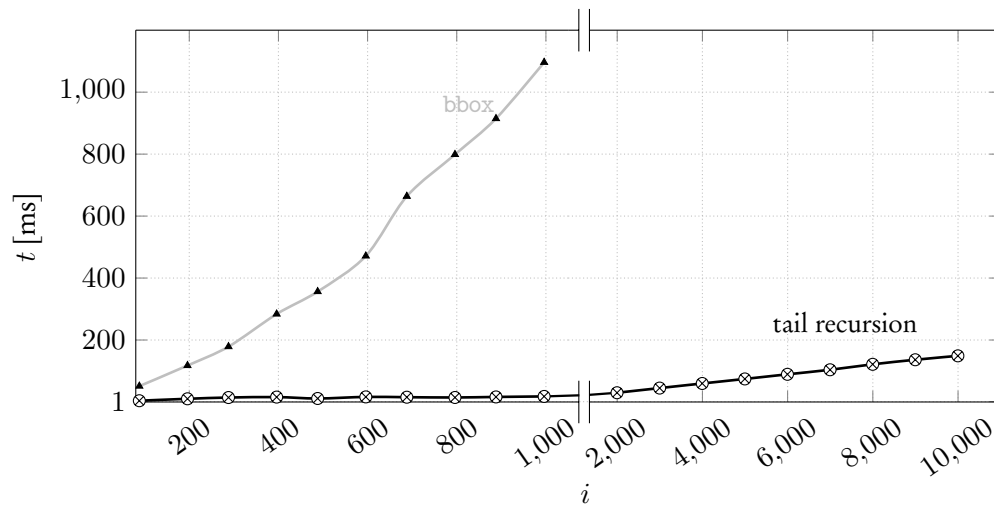


Figure D.6: Evaluating $\text{bbox}(\text{move}(p), p, \text{box}(p, p))$ finds the bounding box of a perfect 2D circle with diameter i beginning at position p located on the edge of the 2D circle.

returns the 2D bounding box of two 2D boxes b_1 and b_2 . Function bbox is recursively defined as:

$$\begin{aligned} \text{bbox}(p_c, p_c, b) &= b \\ \text{bbox}(p_c, p_g, b) &= \text{bbox}(\text{move}(p_c), p_g, b \boxplus \text{box}(p_c, p_c)). \end{aligned} \quad (\text{bbox})$$

Function bbox is tail-recursive. Figure D.6 compares execution times of bbox before and after compilation. Note, that evaluation of bbox before compilation terminates prematurely due to stack overflow at $i = 4,000$ after about 8 seconds.

D.7 MANDELBROT SET FRACTALS

Function $\text{mandel}(0, x_c, y_c, x_c, y_c, p)$ computes the iteration number of the Mandelbrot set at position (x_c, y_c) with precision p . It is recursively defined as:

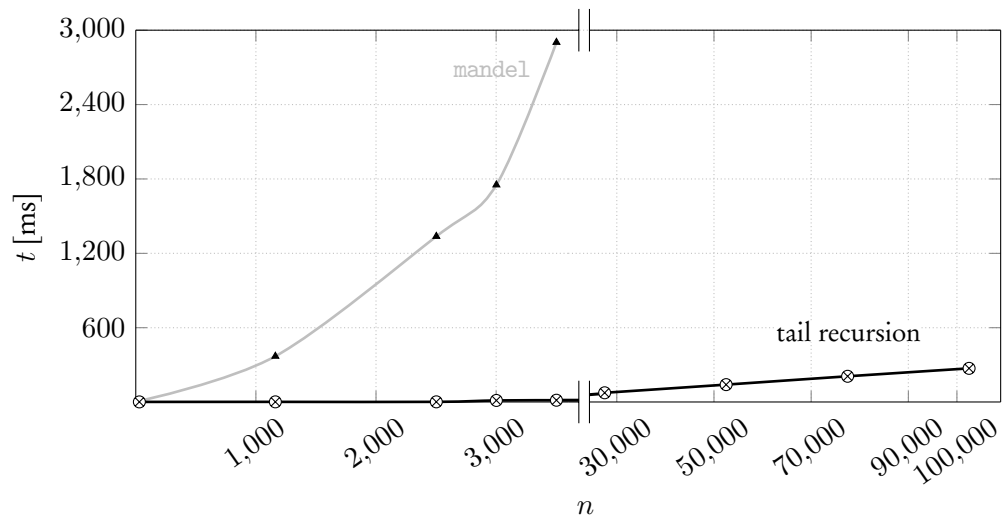


Figure D.7: Evaluating $\text{mandel}(0, x_c, y_c, x_c, y_c, p)$ calculates the iterations of the Mandelbrot set at pixel (x_c, y_c) with precision p requiring n iterations.

$$\text{mandel}(i, x_c, y_c, x, y, i) = i \tag{mandel}$$

$$\text{mandel}(i, x_c, y_c, x, y, p) = \begin{cases} i & , x^2 + y^2 \geq p \\ \text{mandel}(i + 1, x_c, y_c, x^2 - y^2 + x_c, 2xy + y_c, p) & , \text{otherwise.} \end{cases}$$

Function `mandel` is tail-recursive. Figure D.7 compares execution times of `mandel` before and after compilation. Evaluation of `mandel` before compilation terminates prematurely due to stack overflow at $n = 27,502$ after about 1 second.

BIBLIOGRAPHY

- [55] Abelson, H., Sussman, G., & Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. The MIT Press.
- [56] Advanced SQL (2020). Advanced SQL Lecture. <https://db.inf.uni-tuebingen.de/teaching/AdvancedSQLSS2020.html>.
- [57] Aranda, G., Nieva, S., Sáenz-Pérez, F., & Sánchez-Hernández, J. (2013). R-SQL: An SQL Database System with Extended Recursion. *Electronic Communications of the EASST*, 64.
- [58] Barraclough, R., Binkley, D., Danicic, S., Harman, M., Hierons, R., Kiss, A., Laurence, M., & Ouarbya, L. (2010). A Trajectory-Based Strict Semantics for Program Slicing. *Theoretical Computer Science*, 411(11–13).
- [59] Bayer, R. & McCreight, E. (1970). Organization and Maintenance of Large Ordered Indices. SIGFIDET '70 (pp. 107–141). New York, NY, USA: Association for Computing Machinery.
- [60] Berndt, D. & Clifford, J. (1994). Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the KDD Workshop* Seattle, WA, USA.
- [61] Binnig, C., Behrmann, R., Faerber, F., & Riewe, R. (2012). FunSQL: It is Time to Make SQL Functional. In *Proceedings of the EDBT/ICDT DanaC Workshop* Berlin, Germany.
- [62] Bird, R. (1980). Tabulation Techniques for Recursive Programs. *ACM Computing Surveys*, 12(4).
- [63] Blelloch, G. (1996). Programming Parallel Algorithms. *Commun. ACM*, 39(3), 85–97.

- [64] Boehm, M., Kumar, A., & Yang, J. (2019). *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool.
- [65] Burghardt, T., Hirn, D., & Grust, T. (2022). Functional Programming on Top of SQL Engines. In J. Cheney & S. Perri (Eds.), *Practical Aspects of Declarative Languages* (pp. 59–78). Cham: Springer International Publishing.
- [66] Codd, E. (1979). Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.*, 4(4), 397–434.
- [67] Cormen, T., Leiserson, E., Rivest, R., & Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.
- [68] DBWorld (2020). DBWorld Mailing List. <https://dbworld.sigmod.org/> (formerly: <https://research.cs.wisc.edu/dbworld/browse.html>).
- [69] Dragicevic, P. (2015). *HCI Statistics without p-values*. Research Report RR-8738, Inria.
- [70] Duta, C. (2022). Another Way to Implement Complex Computations: Functional-Style SQL UDF. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA '22* New York, NY, USA: Association for Computing Machinery.
- [71] Duta, C. & Grust, T. (2020). Functional-Style SQL UDFs With a Capital 'F'. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20* (pp. 1273–1287). New York, NY, USA: Association for Computing Machinery.
- [72] Duta, C., Hirn, D., & Grust, T. (2020). Compiling PL/SQL Away. In *Proceedings of the 10th CIDR Conference* Amsterdam, The Netherlands.
- [73] Fan, J., Gerald, A., Raj, S., & Patel, J. (2015). The Case Against Specialized Graph Analytics Engines. In *Proceedings of the 7th CIDR Conference* Asilomar, CA, USA.

- [74] Fawaz, H.I. Forestier, G., Weber, J., Idoumghar, L., & Muller, P.-A. (2019). Deep Learning for Time Series Classification: A Review. *Data Mining and Knowledge Discovery*, 33(4).
- [75] Finkelstein, S. J., Mattos, N., Mumick, I., & Pirahesh, H. (1996). Expressing Recursive Queries with SQL. *ISO Technical report*.
- [76] Fischer, T., Hirn, D., & Grust, T. (2022). Snakes on a plan: Compiling python functions into plain sql queries. SIGMOD '22 (pp. 2389–2392). New York, NY, USA: Association for Computing Machinery.
- [77] Fredman, M. (1982). The Complexity of Maintaining an Array and Computing its Partial Sums. *JACM*, 29(1).
- [78] Graham, R., Knuth, D., & O., P. (1994). *Concrete Mathematics: A Foundation for Computer Science, 2nd*. Addison-Wesley.
- [79] Guo, P. & Engler, D. (2011). Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In *Proceedings of the ISSTA Conference Toronto, Canada*.
- [80] Hirn, D. & Grust, T. (2020). PL/SQL Without the PL. In *Proceedings of the 39th SIGMOD Conference Portland, OR, USA*.
- [81] Hirn, D. & Grust, T. (2021). *One WITH RECURSIVE is Worth Many GOTOs*, (pp. 723–735). Association for Computing Machinery: New York, NY, USA.
- [82] Horowitz, E. (1983). *Fundamentals of Programming Languages*. Springer.
- [83] Hudak, P., Hughes, J., Peyton-Jones, S., & Wadler, P. (2007). A History of Haskell: Being Lazy with Class. In *Proceedings of the HOPL III Conference San Diego, CA, USA*.
- [84] Knuth, D. (1998). *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley.

- [85] Libkin, L. (2003). Expressive Power of SQL. *Theor. Comput. Sci.*, 296(3), 379–404.
- [86] Lorensen, W. & Cline, H. (1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87* (pp. 163–169). New York, NY, USA: Association for Computing Machinery.
- [87] Marsh, R. (2022). How to Solve This Issue With CTE? Stack Overflow. <https://stackoverflow.com/q/18789502>.
- [88] Martello, S. & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc.
- [89] Michie, D. (1968). “Memo” Functions and Machine Learning. *Nature*, 218(306).
- [90] MySQL (2022). *MySQL 8.0 Documentation*. <http://dev.mysql.com/doc/>.
- [91] Needleman, S. & Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3), 443–453.
- [92] Neumann, T. (2011). Efficiently Compiling Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*, 4(9).
- [93] Neumann, T. & Freitag, M. (2020). Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the 10th CIDR Conference* Amsterdam, The Netherlands.
- [94] Norvig, P. (1991). Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics*, 17(1).
- [95] Oracle (2022). *Oracle 19c Documentation*. <http://docs.oracle.com/>.
- [96] Passing, L., Then, M., Hubig, N., Lang, H., Schreier, M., Günemann, S., Kemper, A., & Neumann, T. (2017). SQL- and Operator-Centric Data Analytics in Relational Main-Memory Databases. In *Proceedings of the 20th EDBT Conference* Venice, Italy.

- [97] PostgreSQL (2022). *PostgreSQL 13 Documentation*. <http://www.postgresql.org/docs/13/>.
- [98] Rowe, L. & Stonebraker, M. (1987). The POSTGRES Data Model. In *Proceedings of the 13th VLDB Conference* Brighton, UK.
- [99] S., H. & J., H. (1992). Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22), 10915–10919.
- [100] SQL Server (2022). *Microsoft SQL Server 2019 Documentation*. <http://docs.microsoft.com/en-us/sql>.
- [101] SQL:1999 (1999). *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.
- [102] SQLite3 (2022). *SQLite3 Documentation*. <https://www.sqlite.org/appfunc.html>.
- [103] Steele Jr, G. (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.
- [104] Tip, F. (1995). A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3).
- [105] Torczon, L. & Cooper, K. (2007). *Engineering a Compiler*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd edition.
- [106] Ulrich, A. & Grust, T. (2015). The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15 (pp. 1421–1426). New York, NY, USA: Association for Computing Machinery.
- [107] Weiser, M. (1982). Programmer Use Slices When Debugging. *Communications of the ACM*, 25(7).

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