Another Way to Implement Complex Computations: Functional-Style SQL UDF

Christian Duta
Department of Computer Science
University of Tübingen, Germany
christian.duta@uni-tuebingen.de

ABSTRACT

Whenever data-intensive computation gets so complex that it requires the use of iteration or recursion, SQL developers turn towards recursive common table expressions (CTEs). We present the results of a user study that shows how developers struggle with the unusual fixpoint semantics and awkward monolithic syntactic structure of CTEs. The study suggests that recursive user-defined functions (UDFs)—written in a style much like regular functional programs—are less prone to errors, significantly more readable, and can be authored more quickly. Since such recursive UDFs can be automatically compiled into efficiently executable CTEs, we put functional-style UDFs forward as another promising pillar to express complex computation close to the data.

CCS CONCEPTS

• **Human-centered computing** → **User studies**; • **Software and its engineering** → **Functional languages**; • **Information systems** → **Relational database query languages**.

KEYWORDS

Functional Programming, Recursive SQL Queries, User Study

ACM Reference Format:
Christian Duta. 2022. Another Way to Implement Complex Computations: Functional-Style SQL UDF. In *Workshop on Human-In-the-Loop Data Analytics (HILDA '22), June 12, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3546930.3547508

HILDA '22 , June 12, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9442-0/22/06…$15.00
https://doi.org/10.1145/3546930.3547508

```
1  WITH RECURSIVE
2  T(c1,...,cn) AS (
3   q0
4    UNION
5   q○(T)
6  )
7  TABLE T;
```

(a) Recursive CTE.

```
1 u ← DISTINCT(q0)
2 w ← u
3 LOOP
4   i ← DISTINCT(q○(w) ∖ u)
5   IF i = ∅ THEN BREAK
6   u ← u ∪ i
7   w ← i
8 END
9 RETURN u
```

(b) Pseudo code.

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

1 WHERE TO MOVE COMPLEX COMPUTATION

Many developers find themselves in a situation where they need computations performed inside an RDBMS. Many popular RDBMSs provide these developers with SQL dialects that come with a rich toolbox of features to help them implement some of the computations. But, the more complex the computation becomes, say recursive algorithms over tabular data, e.g., graph algorithms, the deeper the developer has to reach into the SQL toolbox. And whenever `SELECT-FROM-WHERE` does not cut it anymore, a developer may have to choose the more expressive *recursive common table expression* (CTE) [12], which many popular RDBMSs support since its introduction in SQL:1999 [17]. However, this usually requires the developer to restructure the algorithm into a formulation that fits the rigid fix-point semantics of recursive CTEs (see Figure 1). Recursive CTEs impose restrictions on the programmer (linearity, monotonicity), possibly moving the implementation further away from its original formulation. Finkelstein et al. argue in favor of the potential benefits these rigid recursive CTEs provide [8], but many RDBMS, like PostgreSQL, implement only the restrictions and none of the benefits, as far as we know [16]. Simple syntactic cover-ups are sufficient to lift these restrictions, allowing even for non-monotonic recursive CTEs to use grouping, aggregates, and anti-joins.

"Is there no other way to move computation close to the data?"

```
CREATE FUNCTION knap(k int, u int)
RETURNS int AS $$
 SELECT CASE
  WHEN k = 1 THEN 0
  ELSE (
   SELECT CASE
    WHEN i.w > u THEN knap(k-1,u)
    ELSE
     GREATEST(knap(k-1,u),
              knap(k-1,u-i.w)+i.p)
   END
   FROM   items AS i
   WHERE  i.i = k
  )
 END;
$$ LANGUAGE SQL;
```

(a) Functional-style UDF

$$knap(1,u) = 0$$

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

(b) Textbook-style

```
CREATE FUNCTION knap(k int, u int)
RETURNS int AS $$
 WITH RECURSIVE sack(i,w,p) AS (
  SELECT 1,0,0
   UNION
  SELECT s.i+1,s.w+c.w,s.p+c.p
  FROM   sack AS s,items AS i,
         LATERAL (
          VALUES (0,0),(i.w,i.p)
         ) AS c(w,p)
  WHERE  s.w+i.w <= u
  AND    i.i = s.i+1
  AND    s.i <= k )
 SELECT MAX(s.p) FROM sack AS s;
$$ LANGUAGE SQL;
```

(c) Recursive CTE

Figure 2: Each function in Figures 2(a) to 2(c) solves the *0-1 Knapsack* problem. A rough comparison of the function body in Figure 2(b) with the body found in Figure 2(a) shows strong similarities when set side by side. However, compared to Figure 2(c), the function bodies look almost unrecognizably different.

Functional-style SQL user-defined functions (we call functional-style UDFs) gives developers another way of expressing complex computations. Functional-style UDFs are your run-of-the-mill SQL functions which allow recursive self-invocations inside their function body.

In this publication, we present the results of a user study to see if functional-style UDFs are a welcome addition to how a developer writes complex computations close to the data. We argue that there are algorithms where developers have an easier time writing and thinking about them in the form of a functional-style UDF when compared to the more restrictive and almost assembly-like recursive CTEs. The cognitive load a developer has to juggle when working with recursive CTEs can be alot more straining when compared to functional-style UDFs.

Consider the *0-1 Knapsack* problem [13], for example. Given items $i \in 1,\ldots,n$ where each item has a weight w_i and a value p_i. Then $knap(n,w)$, recursively defined in its textbook-style formulation in Figure 2(b), maximizes the sum of values of items that fit into a knapsack of weight w. The recursive CTE formulation (Figure 2(c)), while compact, looks very different than its textbook-style form. Compared to that, the functional-style UDF formulation (Figure 2(a)) does not significantly alter the textbook-style form. The user study results in Section 3 discuss wether functional-style UDFs are feasible to become another pillar of support for developers to move their complex computations into RDBMSs. Section 4 summarized the results of this study.

Recursive CTE as Target Language. Some RDBMSs, like PostgreSQL, support functional-style UDFs out-of-the-box. However, they do not support any optimizations that would improve the performance of recursive functional-style UDFs. And it shows. Thus, we defined a SQL-to-SQL compiler [7] which utilizes a wide variety of optimizations. The compiler takes a functional-style UDF body f and replaces it with a semantically equivalent function body f' without self-invocations using recursive CTEs instead. The function f' then evaluates in a two-step process:

(1) **Construct call graph** g top-down. This step records all recursive calls that f would have performed. We utilize the well known *program slicing technique* [20, 18] which enables the function to find all recursive calls without invoking itself recursively. This continues until base cases have been reached.
(2) **Traverse** call graph g bottom up. Start with the base cases and record intermediate results until we produce and return the result of the initial function call at the root of call graph g.

This leaves the doors wide open for optimizations without modifying the underlying RDBMS. Some of them we list here:

- **Sharing.** Collapse all recursive calls with the same arguments into one to remove redundant function call nodes in the call graph.
- **Memoization.** Store the results of the initial and all intermediate recursive function calls inside a memoization table m for later function calls.
- **Linear-/Tail-Recursion.** The compiler detects and exploits if a functional-style UDF is linear- or tail-recursive.

Only when compiling functional-style UDFs do many algorithms become feasible to run inside the RDBMS. This

includes widely used algorithms like *dynamic time warping* [3] (DTW), which have their call graph size reduced significantly from exponential to quadratic size when compiled. Thus, runtime measurements come close to that of a carefully optimized hand-written version of DTW in its pure recursive CTE form.

Further compilation methods for SQL UDFs with self-invocations exist. One notable research effort takes the function body and, using a form of *continuation-passing style transformation*, translates it into a function body without self-invocations [4]. Another, called *R-SQL* [2], translates SQL functions with self-invocation into a set of queries with `SELECT` and `INSERT` statements. An external program, written in Python, evaluates these statements until a fixpoint is reached.

2 MORE RELATED WORK

While this publication focuses strictly on functional-style UDFs and recursive CTEs, there is research effort for ways to give developers further options to express complex algorithms.

Some RDBMS offer a non-standard *procedural* extension to SQL. *PL/SQL* is such an extension. It originated as part of the Oracle database [15] and spawned a derivate *PL/pgSQL* which ships with PostgreSQL [16]. However, the performance of *PL/pgSQL*-functions are disappointing, as each SQL query embedded inside a statement requires a context switch from extension executor to query executor and back. This can be alleviated by translating these functions into their semantically equivalent `LANGUAGE SQL` counterpart [10].

Further research was conducted for *HyPer* [14], extending the SQL language itself by adding an `ITERATE`-operator. This operator allows for an iterative subquery inside a SQL query. The `ITERATE`-operator works similar to a *for*-loop, where the developer defines an initialize-, iterate- and stop-expression.

3 USER STUDY

In 2020, we conducted an anonymous online study which targeted three demographics: students that have attended the *Advanced SQL* lecture of summer semester 2020 [1], users subscribed to the *DBWorld mailing list* [5], and interested students, as well as staff of the University of Tübingen, subscribed to the user study mailing list. Students who 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. The complete user study form is viewable online [19].

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 in response. 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 [6] deems this sufficient to draw meaningful conclusions from. 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.

User study introduction. Before the user study proper, an introductory text informed every participant *not* to use any external programs to run queries found in this study. Instead, they were asked to use pen and paper only, if at all.

Then, we asked them to list three regular programming languages they are 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 those 19 participants form a homogenous group where everyone is familiar with at least one imperative programming language, the majority being Java and Python. And 15 of them had at least some exposure to Functional Programming.

The study itself is segmented into four general topics: choose the correct implementations (Section 3.1), describe a code snippet (Section 3.2), manually evaluate a code snippet (Section 3.3), and write the *0-1 knapsack* algorithm (Section 3.4).

3.1 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 seperated into two tasks, each with its own textbook-style algorithm: *Fibonacci Numbers* [9] (`fib`):

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

and *Greatest Common Divisor* [11] (`gcd`):

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

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`). Then, they are

	Task	Points min	avg	max	Σ	CI (95%)	p-Val
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] min	avg	max	CI (95%)	p-Val
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.

Figure 3: The aggregated scores and times of the tasks *Fibonacci Numbers* (fib) and *Greatest Common Divisor* (gcd).

***Scoring scheme:* participants earn +1 point, if a selection is correct and -1 point, if incorrect.**

tasked to select *only* those snippets that correctly implement the algorithm. The participants submit the time they need to complete this task.

Scores and times of each task `fib` and `gcd` are aggregated in Figures 3(a) and 3(b). The aggregated times for each task show significantly lower times for functional-style UDFs compared to recursive CTEs. One participant skipped both recursive CTE parts of `fib` and `gcd`.

Conclusion. The aggregated scores in Figure 3 suggest that participants perform similarly well in differentiating between correct and incorrect implementations if we compare functional-style UDFs and recursive CTEs. However, participants require about half the time for functional-style UDFs. The reason is that recursive CTEs disfigure the simple formulation of the textbook-style form, and thus it becomes much more time-consuming to detect any errors. Functional-style UDFs, on the other hand, are very similar looking and can be compared almost verbatim to the textbook-style form, which is ideal.

3.2 Describe a Code Snippet

The second topic is also separated into two tasks (*Comprehension I* and *Comprehension II*). It gauges the ability and speed with which each participant understands and correctly describes an undocumented code snippet. One is a functional-style UDF, and the other is a recursive CTE. Both snippets

```
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;
```

Figure 4: Functional-style UDF `f(i,j,k)` of task *Comprehension I.*

```
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;
```

Figure 5: Recursive CTE `g(a)` of task *Comprehension II.*

were chosen for their elegance and concise formulation in their specific style:

- a functional-style UDF `f(i,j,k)` which applies an error that propagates over time (see Figure 4), and
- a recursive CTE `g(a)` which traverses a directed graph and sums up each edge weight it passes once (see Figure 5).

The participants submit the time they need to complete this task. Scores and times of tasks `fib` and `gcd` are aggregated in Figures 6(a) and 6(b). Two participants skipped the recursive CTE in *Comprehension II*.

Conclusion. The aggregated scores in Figure 6(a) show that twice as many participants described the functional-style UDF correctly (i.e., scored two or more points). Participants seem to have difficulty explaining what the recursive CTE does without discovering its intended purpose. The more convoluted nature of recursive CTEs may be the reason (recall Figure 7). The aggregated times in Figure 6(b) suggest that each function took the participants about the same amount of time until they were confident enough to submit their results. Thus, we conclude that developers are more likely to provide an accurate description of an undocumented functional-style UDF than they would an undocumented recursive CTE in roughly the same amount of time.

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.

Figure 6: The aggregated scores and times of tasks *Comprehension I* and *Comprehension II*.

***Scoring scheme:* 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).**

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.

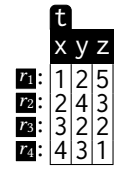
Figure 7: The aggregated scores and times of task *Evaluation*.

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

3.3 Manually Evaluate a Code Snippet

The third topic, *Evaluation*, builds upon the functions of the previous tasks *Comprehension I* and *Comprehension II* in Section 3.2. This topic gauges the ability and speed of the participants to manually evaluate a sample function call `f(1,3,0)` and `g(4)` (recall the functions `f` and `g` in Figures 4 and 5). 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 Figures 7(a) and 7(b). Only the scores differ between functional-style UDF and recursive CTE. Two participants skipped evaluating the function of *Comprehension II*.



(a) Sample table.

1.	**Inititalize** *Line 1,2*		u = ⦇r4⦈	w = ⦇r4⦈
2.	**Iterate** *Line 4,6,7*	i = ⦇r3⦈	u = ⦇r4 r3⦈	w = ⦇r3⦈
3.	**Iterate** *Line 4,6,7*	i = ⦇r2⦈	u = ⦇r4 r3 r2⦈	w = ⦇r2⦈
4.	**Stop** *Line 4,5*	i = ⦇ ⦈	↪ RETURN u	

(b) Bag Variable Trace.

Figure 8: Evaluating a recursive CTE manually requires the developer to keep track of three bag variables `i`, `u`, and `w` (recall Figure 1(b)). The complete variable trace of function `g` (Figure 5) for call `g(4)`. The ⦇ ⦈ denote bags.

We find that participants who did well for the previous tasks in Section 3.2 also performed well in both tasks here. This is unsurprising since both tasks utilize the same functions `f` and `g`. Thus, participants that understood a function were almost certainly going to evaluate them correctly.

Conclusion. 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 and produce the final result.

This stands in stark contrast with recursive CTEs. Developers have to juggle many implicit variables throughout the evaluation process. The complete bag variable trace (see Figure 8) highlights the cognitive load a developer must juggle to evaluate the function call `g(4)` manually.

Both functions `f` and `g` were chosen for their elegant implementation in their respective style. However, these results make it easy to argue in favor of preferring the more explicit and readable nature of functional-style UDFs compared to recursive CTEs and their awkward syntax.

3.4 Write the 0-1 Knapsack Algorithm

This final topic, *0-1 Knapsack*, requires the participants to choose between writing a functional-style UDF or recursive CTE. First, the topic presents the participants with the textbook-style algorithm of the *0-1 Knapsack* problem [13] (recall Figure 2(b)). The participants then submit their implementation and the time they need to complete this task.

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.

Figure 9: The scores and aggregated times of task *0-1 Knapsack*.

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

Scores and times of task *0-1 Knapsack* are aggregated in Figures 9(a) and 9(b). No participants tried to implement 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* (Figure 2(b)) algorithm for this task because both implementations as a functional-style UDF (Figure 2(a)) and recursive CTE (Figure 2(c)) are of similar complexity, which makes them comparable to each other. However, only the functional-style UDF resembles the textbook-style algorithm.

This supports our claim that some algorithms do not lend themselves to be easily rewritten into their equivalent recursive CTE formulation. Thus, if developers have the choice, they may choose functional-style UDFs over recursive CTEs.

4 SUMMARY

Overall, this user study suggests that functional-style UDFs allow developers a feasible alternative to implementing complex computation inside an RDBMS. We found that only tasks which focus on recursive CTEs were skipped. This further hints at a need for alternative ways to write complex computations besides recursive CTEs.

For developers familiar with (functional) programming languages, we found that some functional-style UDFs require about half the time to distinguish correct from incorrect compared to recursive CTEs. Developers are more likely to correctly describe and manually evaluate a functional-style UDF when compared to recursive CTEs. And finally, we are now confident that there exist (textbook-style) algorithms, where developers would prefer writing them as a functional-style UDF instead of a recursive CTE. Thus, we conclude that functional-style UDFs would support developers to move even more complex computations closer to the data.

REFERENCES

[1] Advanced SQL Lecture, 2020. https://db.inf.uni-tuebingen.de/teaching/AdvancedSQLSS2020.html.

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

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

[4] T. Burghardt, D. Hirn, and T. Grust. Functional Programming on Top of SQL Engines. In J. Cheney and S. Perri, editors, *Practical Aspects of Declarative Languages*, pages 59–78, Cham, 2022. Springer International Publishing.

[5] DBWorld Mailing List. https://dbworld.sigmod.org/ (formerly: https://research.cs.wisc.edu/dbworld/browse.html).

[6] P. Dragicevic. HCI Statistics without p-values. Research Report RR-8738, Inria, June 2015.

[7] C. Duta and T. Grust. Functional-Style UDFs With a Capital 'F'. In *Proc. SIGMOD*, Portland, OR, USA, June 2020.

[8] S. J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing Recursive Queries with SQL. *ISO Technical report*, 1996.

[9] R. Graham, D. Knuth, and Patashnik O. *Concrete Mathematics: A Foundation for Computer Science, 2nd.* Addison-Wesley, 1994.

[10] D. Hirn and T. Grust. *One WITH RECURSIVE is Worth Many GOTOs*, page 723–735. Association for Computing Machinery, New York, NY, USA, 2021.

[11] D. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[12] L. Libkin. Expressive Power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, mar 2003.

[13] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, Inc., USA, 1990.

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

[15] *Oracle 19c Documentation.* http://docs.oracle.com/.

[16] *PostgreSQL 14 Documentation.* http://www.postgresql.org/docs/14/.

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

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

[19] Recursion in SQL - User Study. https://recursion-in-sql.github.io/user-study/.

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