

New Compilation Methods for Complex User-Defined Functions

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Denis Hirn
aus Tübingen

Tübingen
2024

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

Tag der mündlichen Qualifikation:	19.06.2024
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Torsten Grust
2. Berichterstatter:	Prof. Dr. Hannes Mühleisen
3. Berichterstatter:	Ass. Prof. Dr. Amir Shaikhha

“If I have seen further it is by standing on the shoulders of Giants.”

—Sir Isaac Newton

Acknowledgements

I am extremely proud of the work that we have been able to accomplish during my time as a PhD student under the supervision of *Torsten Grust*. Torsten has been an incredible mentor, that has always been supportive and encouraging. While he typically had some ideas on where to go next, he was also the kind of supervisor that let me explore my own ideas, because he knew that eventually, some of them would turn out to be good ones. I am grateful for the trust that he has put in me, and for the freedom that he has given me to explore and to grow as a researcher. Thank you, Torsten!

I would also like to thank *Hannes Mühleisen* for agreeing to be a reviewer of this dissertation. I did not take this for granted, and I am grateful for the time that he has spent reading and reviewing my work. Having him review it was a huge motivational push for me to get it done and to polish it to the best of my ability.

Also, I want to thank *Amir Shaikhha*, who also agreed to be a reviewer of this dissertation. This is nothing that I take lightly, and I am delighted that he has agreed to spend his time reading and reviewing my work.

Last but not least, I would like to thank the members of the dissertation committee for letting me defend this dissertation and approve me for the PhD degree.

Research is teamwork, and without my colleagues, this dissertation would be a very different one. I would like to thank *Tim Fischer*, who has been an immense help during the writeup of this work. I knew that I could always lend myself some brainpower from him, and he always had a good idea or two to share. He motivated me to *just do the work*, and to dig deeper into some remaining problems that I had not yet solved. This eventually led to some of the results that are presented in this dissertation. Also, I would like to thank *Louisa Lambrecht* who did help me with proofreading this dissertation.

Finally, I would like to thank my family and friends for their support and encouragement throughout my time as a PhD student. I am especially thankful for the patience and understanding that *Anna* has shown me during this time. I am thankful for the opportunities that I have been given, and for the people that have been part of this journey with me.

Thank you all!

Abstract

User-defined functions (UDFs) can be used to extend database systems with custom functionality. However, the performance of UDFs is often disappointing, which renders them useless for many applications. It has become common developer wisdom to avoid UDFs in performance-critical applications if possible. Previous work on UDF performance improvements has left many insights by the programming languages community unexplored. This thesis aims to fill this gap by leveraging programming language techniques. We show that these techniques can be adapted such that SQL can be used to express both *iterative* and *recursive* UDFs in a way that improves performance over the initial UDF formulation.

In the first part of this thesis, we will focus on *iterative* UDFs written in variants of PL/SQL. The imperative statement-by-statement style of programming clashes with the plan-based evaluation of SQL queries resulting in an *impedance mismatch* and therefore friction at runtime, slowing PL/SQL execution down significantly. State of the art approaches can only compile UDFs with simple, linear control flow and cannot handle looping control flow at all, or they require database extensions, which are not portable and not available on all systems. We present a novel approach to UDF compilation that uses *trampolined style* to compile UDFs with arbitrarily complex looping control flow to pure SQL queries. After compilation, the PL/SQL interpreter is no longer needed. The entire computation is expressed in SQL and can be executed without switching back and forth between the SQL executor and PL/SQL interpreter. This solves the impedance mismatch and eliminates the friction during execution. It also allows the database system to optimize the computation as a whole. Since the computation is expressed in SQL, it can be executed on any system that supports contemporary SQL. This can be used to bring UDFs to systems that lack native support. We show how this compilation approach can be used to improve performance, and the planner's optimization capabilities. We present a collection of 18 UDFs that we use to evaluate our approach. In our experiments, we show that our approach can improve performance by up to 3 times. We also show cases where our approach does not improve, or even degrades performance. We discuss the reasons for this and show how to avoid these pitfalls.

The second part of this thesis focuses on *recursive* UDFs. Performance of such recursive UDFs is often disastrous, because database systems are not optimized for these kinds of computations. We use PostgreSQL as an example to show why recursive UDFs are slow, and why the system spends most of the runtime on parsing and planning. This repeated overhead penalizes every function call at runtime, rendering programming with recursive UDFs largely impractical. We propose to take recursive UDFs for what they are: *functions*. Using tried and tested techniques from the field of programming languages, we show how to compile recursive UDFs to pure SQL queries. We reuse large parts of the compilation pipeline from the first part of this thesis, and show that it can be used to compile recursive UDFs as well. Trampolined style will again be the

vehicle for expressing the computation in SQL. We use a set of 10 recursive UDFs to evaluate our approach. Our experiments show that these UDFs typically suffer from over 90% overhead. Compilation eliminates this overhead and improves performance by up to 180 times. This shows that even though functions are not first-class in SQL, they can still be efficient.

The significance of this thesis lies in its identification of trampolined-style SQL as a powerful tool for expressing computations in SQL. It shows that SQL is expressive enough for arbitrarily complex computations, and that it achieves excellent performance. The detailed set of rules presented in this thesis can be used to implement a compiler for UDFs in any database system that supports `LATERAL` joins and recursive CTEs.

Zusammenfassung

User-defined functions (UDFs) können verwendet werden, um Datenbanksysteme um benutzerdefinierte Funktionen zu erweitern. Die Performance von UDFs ist jedoch oft enttäuschend, was sie für viele Anwendungen unbrauchbar macht. Es ist inzwischen eine verbreitete Entwicklerweisheit, UDFs in performancekritischen Anwendungen zu vermeiden, wenn möglich. Frühere Arbeiten zur Verbesserung der UDF-Performance haben viele Erkenntnisse der Programmiersprachen-Community unberücksichtigt gelassen. Diese Arbeit zielt darauf ab, diese Lücke zu schließen, indem Programmiersprachentechniken eingesetzt werden. Wir zeigen, dass diese Techniken so angepasst werden können, dass SQL verwendet werden kann, um sowohl *iterative* als auch *rekursive* UDFs in einer Weise auszudrücken, die die Performance gegenüber der ursprünglichen UDF-Formulierung verbessert.

Im ersten Teil dieser Arbeit konzentrieren wir uns auf *iterative* UDFs, die in Varianten von PL/SQL implementiert sind. Der imperative Programmierstil von PL/SQL steht im Widerspruch zur planbasierten Auswertung von SQL-Abfragen, was zu einem *Impedanzmismatch* und damit zu Problemen während der Laufzeit führt und die Ausführung von PL/SQL erheblich verlangsamt. State-of-the-Art-Ansätze können nur UDFs mit einfachem, linearem Kontrollfluss kompilieren und können entweder keinen Schleifenkontrollfluss verarbeiten, oder sie erfordern Datenbankerweiterungen, die nicht portabel beziehungsweise nicht auf allen Systemen verfügbar sind. Wir stellen einen neuartigen Ansatz zur UDF-Kompilierung vor, der *trampolined style* verwendet, um UDFs mit beliebig komplexem Kontrollfluss in reine SQL-Abfragen zu kompilieren. Nach der Kompilierung wird der PL/SQL-Interpreter nicht mehr benötigt. Die gesamte Berechnung wird mittels SQL ausgedrückt und kann ohne ständiges Hin- und Herwechseln zwischen dem SQL-Executor und dem PL/SQL-Interpreter ausgeführt werden. Dies löst den Impedanzmismatch auf und beseitigt die Probleme zur Laufzeit. Es erlaubt dem Datenbanksystem auch, die Berechnung als Ganzes zu optimieren. Da die Berechnung in SQL ausgedrückt wird, kann sie

auf jedem System ausgeführt werden, das zeitgemäßes SQL unterstützt. Dadurch können UDFs auf Systeme gebracht werden, die keine native Unterstützung bieten. Wir zeigen, dass dieser Kompilierungsansatz verwendet werden kann, um die Performance und die Optimierungsmöglichkeiten des Planers zu verbessern. Wir präsentieren eine Sammlung von 18 UDFs, die wir zur Evaluierung unseres Ansatzes verwenden. In unseren Experimenten zeigen wir, dass unser Ansatz die Performance um bis zu Faktor 3 verbessern kann. Wir zeigen auch Fälle, in denen unser Ansatz die Performance nicht verbessert oder sogar verschlechtert und erörtern die Gründe dafür und zeigen, wie man diese Fallstricke vermeiden kann.

Der zweite Teil dieser Arbeit konzentriert sich auf *rekursive* UDFs. Die Performance solcher rekursiven UDFs ist oft katastrophal, da Datenbanksysteme nicht für diese Art von Berechnungen optimiert sind. Wir verwenden PostgreSQL als Beispiel, um zu zeigen, warum rekursive UDFs langsam sind und warum das System den Großteil der Laufzeit für das Parsen und Planen aufwendet. Dieser wiederholte Overhead bestraft jeden Funktionsaufruf zur Laufzeit und macht das Programmieren mit rekursiven UDFs weitgehend unpraktikabel. Wir schlagen vor, rekursive UDFs als das zu betrachten, was sie sind: *Funktionen*. Mit bewährten Techniken aus dem Bereich der Programmiersprachen zeigen wir, wie man rekursive UDFs in reine SQL-Abfragen kompilieren kann. Dabei verwenden wir große Teile der Kompilierungspipeline aus dem ersten Teil dieser Arbeit wieder und zeigen, dass sie auch zur Kompilierung rekursiver UDFs verwendet werden kann. Trampolined style wird wieder das Mittel sein, um die Berechnung in SQL auszudrücken. Wir verwenden einen Satz von 10 rekursiven UDFs, um unseren Ansatz zu evaluieren. Unsere Experimente zeigen, dass diese UDFs typischerweise unter einem Overhead von über 90% leiden. Die Kompilierung eliminiert diesen Overhead und verbessert die Performance um bis zu Faktor 180. Dies zeigt, dass Funktionen in SQL zwar nicht von *erster Klasse* sind, aber dennoch effizient sein können.

Die Signifikanz dieser Arbeit liegt darin, dass sie trampolined-style SQL als ein leistungsfähiges Werkzeug zum Ausdruck von Berechnungen in SQL identifiziert. Sie zeigt, dass SQL ausdrucksstark genug ist, um beliebig komplexe Berechnungen auszudrücken, und dass eine hervorragende Performance erreicht werden kann. Das detailliert beschriebene Regelwerk der Kompilierung kann verwendet werden, um einen Compiler für UDFs in jedem Datenbanksystem zu implementieren, sofern LATERAL Joins und rekursive CTEs unterstützt werden.

Contents

- Contents** **ix**

- 1. Introduction** **1**
 - 1.1. Thesis Overview and Contributions 4
 - 1.1.1. Compiling PL/SQL Away 4
 - 1.1.2. Functional Programming on Top of SQL Engines 7
 - 1.2. Structure of the Thesis 9

- COMPILING PL/SQL AWAY** **11**

- 2. Avoid PL/SQL if you can...** **13**
 - 2.1. Context Switching 13
 - 2.2. Drawbacks of PL/SQL Evaluation 16
 - 2.3. Behind the Scenes of PL/SQL 17
 - 2.3.1. Embedded SQL 17
 - 2.4. Case Study: UDF route 19

- 3. One Way to Trade PL/SQL for SQL** **23**
 - 3.1. The Expressive Power of SQL 24
 - 3.2. The PL/SQL Language 26
 - 3.3. PL/SQL Control Flow in Terms of GOTO 29
 - 3.4. Tail Recursion Replaces GOTO 31
 - 3.5. Trampoline Style Tames Mutual Recursion 34
 - 3.6. Trampoline Style in SQL 36

- 4. Trampoline Style Manages Control, and Data Flow, Too** **39**
 - 4.1. From Scalar Values To Tables 39
 - 4.2. Control Flow Management 40
 - 4.3. Data Flow Management 41
 - 4.4. The Impact of Data Rows in Trampoline Style SQL 42

- 5. From PL/SQL to SQL: Behind the Scenes** **45**
 - 5.1. From PL/SQL to SSA 45
 - 5.2. From SSA to ANF 47
 - 5.3. From ANF to Trampoline Style ANF 49
 - 5.4. From Trampoline Style ANF to SQL 50

- 6. Experiments** **53**
 - 6.1. Compiling a Collection of UDFs 53
 - 6.2. PostgreSQL 11 vs. 15—What has changed? 56
 - 6.2.1. Cost-based Optimization 58
 - 6.3. To Recurse is Divine, to ITERATE Space-Saving 59

- 7. Related Work and Conclusion** **63**
 - 7.1. Conclusions 63
 - 7.2. Related Work: Froid 65
 - 7.3. Related Work: Aggify 66

7.4. Related Work: ByePy	67
7.5. Related Work: User-Defined Operators	67
7.6. Related Work: Tupleware	68
7.7. More Related Work	69
FUNCTIONAL PROGRAMMING ON TOP OF SQL ENGINES	71
8. Recursive SQL UDFs	73
8.1. From 1000s of Plans to <i>One</i> Plan	73
9. Treating Recursive UDFs Like Functions	77
9.1. Translation from SQL to SSA _{REC}	78
9.2. Transition to ANF _{REC}	80
9.3. From Recursion Towards Iteration: CPS and Defunctionalization	81
9.4. Trampoline Style: A Single Loop Replaces Mutual Recursion	85
9.5. Memoizing the Results of Recursive Calls	88
9.6. Implementation of Continuation Stacks	89
10. Experiments	91
11. From Recursion to Iteration to SQL—Marching Squares	95
11.1. Recursive Marching Squares	95
11.2. Recursive PL/SQL	96
11.3. Experiments	99
12. Conclusion and Related Work	103
12.1. Conclusions	103
12.2. Related Work: Functional-Style SQL UDFs	104
12.3. Related Work: First-Class Functions for First-Order Database Engines	105
12.4. Related Work: Fun SQL	105
12.5. Incrementalization	106
FINAL REMARKS	107
13. Wrap-Up	109
13.1. Future Work	110
APPENDIX	113
A. PL/SQL UDF Definitions	115
Function <code>bbox</code>	115
Function <code>global</code>	115
Function <code>force</code>	116
Function <code>items</code>	116
Function <code>late</code>	117
Function <code>margin</code>	117
Function <code>markov</code>	118
Function <code>packing</code>	119
Function <code>savings</code>	120

Function <code>sched</code> .	121
Function <code>service</code> .	121
Function <code>sheet</code> .	122
Function <code>ship</code> .	123
Function <code>sight</code> .	123
Function <code>visible</code> .	124
Function <code>vm</code> .	125
Function <code>ray</code> .	126
B. Recursive UDF Definitions	127
Function <code>comps</code> .	127
Function <code>dtw</code> .	127
Function <code>eval</code> .	127
Function <code>fsm</code> .	128
Function <code>lcs</code> .	128
Function <code>mbrot</code> .	128
Function <code>paths</code> .	128
Function <code>vm</code> .	129
Bibliography	131
List of Terms	139

Introduction

1.

Modern SQL¹ is a domain-specific programming language that is both *declarative* and *Turing-complete*. Today, it has all the features needed to handle analytical and transactional workloads. Each new revision of the SQL standard extends and improves the language's feature set, which in turn improves its usability and usefulness in various applications. During the early days of SQL—particularly prior to SQL:1999—this was not the case. Since its beginnings in 1974, SQL has evolved as *the* query language of relational database management systems. Chamberlin and Boyce's work on "*SEQUEL: A Structured English Query Language*" [1] laid the foundation for today's SQL. The expressive power of SEQUEL corresponded to the first order predicate calculus, which was sufficient at first. However, with the increasing use of relational databases, applications require more complex programming. SQL ultimately failed to meet the needs of developers at the time, because it was not possible to (1) express complex formulas, (2) use procedural logic, or (3) execute a batch of multiple queries [2].

Even though SQL is Turing-complete since SQL:1999 and can—at least in theory—be used to express any computation, many developers are educated and biased to think of imperative solutions to programming problems first. Instead of using SQL, computations are expressed in other ways. In practice, this can negatively affect performance. As is common with declarative languages, SQL queries specify *what* to compute, but not *how* to implement that computation. Because of this property, SQL alone is not powerful enough—or rather, not designed—to express consecutive or imperative statements. To overcome these limitations, database-backed applications are often implemented outside the database system, *e.g.*, using imperative programming languages.

This typically results in an access pattern in which the database system must process one SQL statement after another, resulting in *context switching* overhead. This is especially bad in a networked environment, where every SQL statement issued generates network traffic and is subject to network latency. The parts involved in such a setup are visualized in Figure 1.1. Database applications typically send SQL queries to the DBMS, which performs the computation and returns the result. Regardless of the communication channel, these data transmissions take time. Data transfer latency and context switching overhead can be a major drawback for applications that require fast data access, and fast response times, as both can be compromised. This latency and overhead may become a bottleneck.

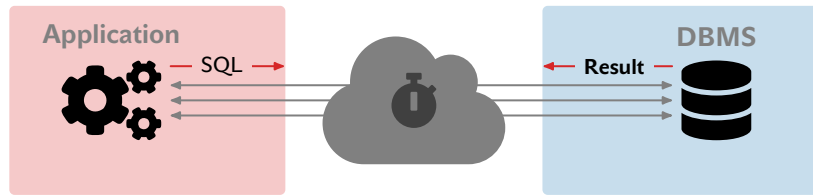
A number of technologies are available to simplify in-database programming and improve the overall quality of life. For example, user-defined functions (UDFs) have several advantages over using external database operations, including the ability to express

1: I consider SQL:1999 and newer as *modern*.

[1]: Chamberlin et al. (1974), 'SEQUEL: A Structured English Query Language'

[2]: Feuerstein et al. (2014), *Oracle PL/SQL Programming*

Figure 1.1: Database applications usually communicate with the DBMS using SQL queries. This communication, however, takes time \bar{t} and can slow down the application considerably, depending on the communication channel.



complex calculations directly in the database. UDFs allow to encapsulate frequently used logic into reusable functions which reduces code duplication and simplifies maintenance because the function is stored in a single location. Updates to this function can be made as needed and will have an effect on all queries that use this function. In addition, UDFs provide an abstraction layer that hides implementation details. Overall, UDFs can help to keep computations in the database, which can reduce the need to transfer data between the database server and its clients. Many database systems support different languages for UDFs, allowing functions to be written in a familiar programming paradigm. This can improve productivity and reduces the time it takes to implement a solution.

The disparity between imperative and “database-friendly” declarative SQL solutions is known as *impedance mismatch*. The literature describes two common categories: (1) A *conceptual impedance mismatch* occurs, when two programming languages that support different programming paradigms are used jointly to implement an application. For example, SQL and a strictly procedural language would conflict with each other, because it is not possible to execute both languages using the same execution strategy. (2) A *structural impedance mismatch* exists when the used languages support different data types, which means that data is represented differently. This can make it hard to exchange data between the two languages. For example, queries typically return sets of tuples. This means, that there must be a mechanism to loop over the query result and a way to extract and convert attributes from the relational structure to a usable data structure in the programming language [3].

[3]: Copeland et al. (1984), ‘Making smalltalk a database system’

[4]: Oracle 19c PL/SQL Documentation

2: To put this into perspective, PL/SQL emerged about two decades before the first database systems featured Turing-complete SQL in the form of `WITH RECURSIVE`.

PL/SQL. The “*Procedural Language extension to the Structured Query Language*” (PL/SQL) [4] made its debut in 1988 with ORACLE’s RDBMS version 6. It is a high-level procedural programming language that allows developers to write custom functions, operators, and algorithms that are not supported by the built-in functions of the database system. The principle idea was to solve SQL’s shortcomings regarding procedural logic. The introduction of this technology was a huge step forward for in-database computing, because SQL’s capabilities were not up to today’s standards². Instead of sending a single SQL statement at a time, PL/SQL made it possible to send imperative programs with embedded SQL statements to the database system (see Listing 1.1, for example). This allows developers to move the computation from the application, *i. e.*, far away from the data, directly into the database server and therefore close to the data it operates on, which is generally recommended. This integration eliminates network traffic and allows the database server’s resources to be used for complex operations, improving performance. PL/SQL

effectively behaves like a database application running inside the DBMS.

PL/SQL predates many conveniences and technologies used in modern development stacks. It can run on any platform where the database system can run, effectively rendering the language platform-independent. Back in the day, this was a huge advantage and selling point, because it enabled and simplified development of in-database computation. This saved time during development. PL/SQL was a role model and has been adopted by many database systems, some of which are listed in Figure 1.2.

While this approach can improve the performance compared to database-external solutions, it does not solve the fundamental context switching problem. This is due to the way PL/SQL is built. PL/SQL is an interpreted language, implemented on top of SQL engines. The imperative, non-SQL statements are interpreted by the PL/SQL subsystem, while all embedded SQL queries are sent to the SQL executor. This means that PL/SQL runs in a separate context from the SQL queries. SQL queries run in context of the SQL engine, and PL/SQL code runs in context of the PL/SQL subsystem. Because these execution contexts are completely disparate, each switch from one context to the other (and back) takes time and therefore causes context switching overhead. It is important to note, however, that it is generally not possible to merge the two execution contexts without bridging the gap between imperative and declarative programming. Finding a solution to this problem is one of the challenges of this thesis.

Recursive User-Defined Functions. Recursive formulations of algorithms are often easy to understand and implement. However, although SQL database systems typically support UDFs, programming recursively with these functions is rarely recommended. The plan-based execution strategy of these systems penalizes each function call at runtime. This is because recursive UDFs can have a significant negative impact on performance and can even cause the query execution to fail due to memory exhaustion. Database systems are not designed to handle recursive UDF computations efficiently. However, the programming language community has been developing techniques to do this for decades. This thesis explores how these techniques can be applied to SQL.

Listing 1.1: A prototypical SQL script with consecutive statements.

```
SQL
IF ... THEN
  SQL
ELSE
  SQL
END IF;
SQL
```

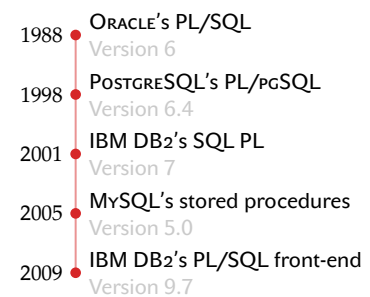


Figure 1.2.: This is a rough timeline of PL/SQL implementations in major database systems.

1.1. Thesis Overview and Contributions

Throughout this thesis, we will distinguish between two classes of in-database functions, namely (1) imperative PL/SQL functions, and (2) recursive UDFs. Both classes of UDFs raise performance issues and can wreck query execution times. The main focus of this work is to explore new compilation methods for these complex UDFs, using proven and tested techniques developed by the programming language community. The goal is to improve the performance of these functions and to make them more usable.

To do this, we use common *intermediate representations (IRs)* used by compilers for imperative, and functional programming languages. These IRs have well-studied correspondences to each other. Therefore, for some of these IRs it is possible to transform one into another if certain constraints are met. Relying on well-established techniques and transformations avoids reinventing the wheel, which makes compilation “easier” and more trustworthy. However, since none of the techniques were developed with SQL as an application in mind, some tweaking is required.

We will use these techniques to develop compilation pipelines capable of transforming UDFs into a single query. This eliminates the context switching overhead for PL/SQL functions, and removes SQL engine-induced inefficiencies for recursive UDFs. For both classes of UDFs, we will show that compilation can significantly improve performance. To determine effectiveness, we will use sets of UDFs and measure important runtime metrics (*e.g.*, speedup factors, memory consumption, and context switches).

This thesis is divided into parts ‘Compiling PL/SQL Away’, and ‘Functional Programming on Top of SQL Engines’, to reflect the two different classes of UDFs we are dealing with. Each part will establish the necessary theory and terminology right before these concepts are used. In Chapter 13, we will summarize the main finding of this thesis and discuss open questions and future work. The following sections provide a brief overview of the two parts of this thesis, and the contributions made in each part.

1.1.1. Compiling PL/SQL Away

RDBMSs are experts in the *plan-based* execution of SQL queries, which is in stark contrast to the imperative evaluation that PL/SQL provides. Even though computations expressed with PL/SQL take place entirely within the database system, the context switching overhead can still become significant. Also, PL/SQL interferes with the database system’s ability to optimize queries, because these functions are like *black-boxes* to the system. This makes it hard to determine cardinalities, prevents global optimizations, and generally means that the system’s optimizer will not be able to generate an optimal execution plan. This can be a major factor in causing suboptimal performance when using PL/SQL.

In this part of the thesis, we describe a compiler that takes PL/SQL UDFs as input and produces semantically equivalent plain (recursive) SQL queries. The compilation pipeline uses the following intermediate program representations:

Static single assignment form (SSA). Programs in SSA express all control flow in terms of *basic blocks* and GOTO statements. SSA has been well studied and is widely used as an IR in modern compilers [5, 6]. The properties associated with programs in SSA form enable many program optimization techniques, such as *dead code elimination*, and *constant propagation*.

Administrative normal form (ANF). ANF is a *direct-style* program representation typically used in compilers for *functional* programming languages such as HASKELL’s GHC [7, 8]. Control flow is expressed in terms of functions and tail-recursive function calls. To bridge the gap between SSA and ANF, we will use a formal mapping described by Chakravarty, Keller, and Zadarnowski.

Trampoline Style. A program in trampoline style is organized as a single “scheduler” loop, the so-called TRAMPOLINE, which manages all control flow. Execution of such programs proceeds in discrete steps. After each step, control and the remaining work are returned to the TRAMPOLINE, which then proceeds to transfer control again [10]. This cycle continues until the program execution is finished. Using TRAMPOLINING, the program is effectively transformed into a state machine. This technique can be used for *tail call elimination* [11, 12]. However, we exploit the property, that only a single loop is required to express arbitrary control flow. This restricted form of control flow perfectly matches the semantics of SQL’s WITH RECURSIVE construct.

Compiling PL/SQL to recursive SQL solves the conceptual impedance mismatch of PL/SQL. Once compiled, there will be no more friction, because the execution will be performed entirely within the SQL context. Switching back and forth between PL/SQL’s interpreter and the SQL executor is no longer necessary, which improves performance. A goal of this effort is to reinvent as few techniques as possible. Therefore, this compiler consists of techniques developed by the programming language community, adapted only when necessary to make it work in the context of SQL.

PL/SQL is the name ORACLE coined for their programming language. However, there is a whole family of imperative in-database programming languages, *i. e.*, ORACLE’s PL/SQL, POSTGRESQL’s PL/PgSQL, or MICROSOFT’s T-SQL. Throughout this thesis, we will almost exclusively use PL/SQL or PL/PgSQL to refer to these languages. However, the principle idea of compiling iterative control flow in terms of recursive CTEs works for all of these languages.

Once PL/SQL is gone, we are left with a recursive SQL query. As the PL/SQL interpreter is no longer required for execution, the query can be executed on any database system featuring a contemporary SQL dialect. Thereby enabling imperative PL/SQL style programmability of the SQL engine, without the need to implement

[5]: Novillo (2003), ‘Tree SSA a new optimization infrastructure for GCC’

[6]: Lattner (2002), ‘LLVM: An infrastructure for multi-stage optimization’

[7]: Maurer et al. (2017), ‘Administrative normal form, continued’

[8]: Maurer et al. (2017), ‘Compiling without Continuations’

[10]: Ganz et al. (1999), ‘Trampoline style’

[11]: Tarditi et al. (1992), ‘No Assembly Required: Compiling Standard ML to C’

[12]: Schinz et al. (2001), ‘Tail call elimination on the Java Virtual Machine’

a PL/SQL interpreter. This allows developers to use the programming paradigm they are most familiar with and still benefit from the power of a SQL engine.

Related Publications. The first part of this thesis is based on the following articles:

Christian Duta, Denis Hirn, and Torsten Grust. ‘Compiling PL/SQL Away’. In: *Proc. CIDR*. 2020

Denis Hirn and Torsten Grust. ‘PL/SQL Without the PL’. In: *Proc. SIGMOD*. 2020

Denis Hirn and Torsten Grust. ‘One WITH RECURSIVE is Worth Many GOTOs’. In: *Proc. SIGMOD*. 2021

Denis Hirn and Torsten Grust. ‘A Fix for the Fixation on Fixpoints’. In: *Proc. CIDR*. 2023

Denis Hirn. ‘Data is Data and Control Should be Data, Too’. In: *Proc. VLDB*. 2023

Contributions.

- ▶ We present a brief overview of the key elements of PL/SQL. This includes control flow statements, table-valued functions, and the SQL engine’s interpreted execution model for PL/SQL (Sections 2.3 and 3.2 and Chapter 2).
- ▶ We formally describe all IRs and transformation steps required to compile PL/SQL functions into pure SQL queries (Chapter 5). Not all PL/SQL functions can be compiled into recursive SQL, therefore we also describe which characteristics a function needs to fulfill to be eligible for this transformation (Section 3.2).
- ▶ We will identify, where context switching occurs during the execution of PL/SQL, and determine the performance impact (Sections 2.3 and 2.4). This will be done using PostgreSQL’s variant PL/pgSQL of PL/SQL (Section 2.1). We will show that the compilation eliminates context switching overhead.
- ▶ We will show that the compilation pipeline is non-invasive, and that the resulting CTEs can be evaluated efficiently by SQL engines. This means that the compilation does not interfere with the database system’s ability to optimize queries (Chapter 6).
- ▶ We will describe a set of PL/pgSQL functions and their compiled SQL counterparts to determine how much performance gain can be expected from compilation (Table 6.1 in Chapter 6). We will also motivate why table-valued functions are particularly well suited for this technique (Chapter 4).
- ▶ Based on the results of these experiments, we will deduce characteristics of PL/SQL functions, to determine when compiling to SQL is likely to improve performance, and when the compiling may degrade performance (Sections 4.1, 6.1 and 7.1).

1.1.2. Functional Programming on Top of SQL Engines

Complex in-database computation can be expressed in many ways. One is functional programming, using *recursive* SQL UDFs. However, this style of programming is rarely supported by SQL database systems and therefore results in abysmal performance characteristics. SQL engines are not optimized for this kind of computation. We consider this a real loss for in-database computation, because functional programming and recursion in general often allow programs to be written in a very concise and elegant way. This can make it easier to understand and maintain these programs.

In this part of the thesis, we will treat recursive SQL UDFs for what they are: **functions**. This allows us to construct a pipeline of function translation techniques that have been well-established and battle-tested by the programming language community. The combination of *continuation-passing-style* (CPS), *defunctionalization*, and *trampolined style* forms the foundation for a non-invasive SQL-level compiler for recursive SQL UDFs. The compilation chain generates a recursive CTE that can be evaluated efficiently by SQL engines. This allows functional programming *close to the data* to still be efficient, even though functions are not first class in SQL. Compared to the current state of the art, which is to avoid recursive UDFs altogether, this is a significant improvement.

Functional representation of UDFs. We will describe a *first-order* functional IR with embedded SQL expressions for recursive SQL UDFs. These expressions are wrapped in “black boxes”, and are not affected by the following compilation steps. The contained SQL fragments are only unwrapped once the final CTE is created.

Continuation passing style (CPS). Since our target is a single-loop interpreter that does not perform any recursive calls, we will rewrite the function into CPS. All functions in CPS take an additional parameter which represents “the rest of the computation”, that is the continuation. All calls are therefore tail-calls, and the computation is effectively sequentialized [18]. Programs in CPS are generally *higher order*.

[18]: Danvy (1994), ‘Back to direct style’

Defunctionalization. Higher-order functional programs can be converted to first-order programs using *Reynold’s* defunctionalization technique [19, 20]. We use defunctionalization in preparation for the compilation back to SQL, where functions are not first class. Applying this technique to the continuations allows us to represent these in terms of data. This is required to proceed to the final step. The resulting program is *first-order* and *mutually tail-recursive*.

[19]: Reynolds (1972), ‘Definitional Interpreters for Higher-Order Programming Languages’

[20]: Danvy et al. (2001), ‘Defunctionalization at Work’

Trampolined Style. We will use *trampolined style* to transform the mutually tail-recursive program into a single loop. This is the same technique we used in the PL/SQL compilation pipeline and we will be able to reuse most of the machinery. We will again target SQL’s **WITH RECURSIVE** construct as the main driver for the function execution.

Related Publications. The second part of this thesis is based on the following article:

Tobias Burghardt, Denis Hirn, and Torsten Grust. ‘Functional Programming on Top of SQL Engines’. In: *Practical Aspects of Declarative Languages: 24th International Symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings*. Philadelphia, PA, USA: Springer-Verlag, 2022. ISBN: 978-3-030-94478-0

Contributions.

- ▶ We will describe a functional IR with embedded SQL expressions for recursive SQL UDFs (Figure 9.2). The SQL expressions are wrapped inside “black boxes”. This allows us to apply our function translation pipeline without the need to consider any SQL specifics.
- ▶ We will formally define all IRs and translation rules required for the compilation (Chapter 9). Parts of this pipeline can be reused from the PL/SQL compilation pipeline (Section 9.4).
- ▶ We will show that the compilation pipeline is non-invasive, and that the resulting CTEs can be evaluated efficiently by SQL engines (Chapter 10).
- ▶ Based on a collection of 10 UDFs, we determine the performance gains this compilation achieves (Table 10.1). We will show that compilation eliminates the overhead introduced by the SQL engine’s terrible execution model for recursive UDFs (Section 9.4).
- ▶ We will show that the compilation technique can be applied to other recursive UDF languages such as PL/SQL. We will also motivate why table-valued functions are particularly well suited for this technique (Chapter 11).
- ▶ Finally we will highlight the benefits of this approach and also discuss the limitations and drawbacks. We will also motivate future research directions (Chapter 12).

1.2. Structure of the Thesis

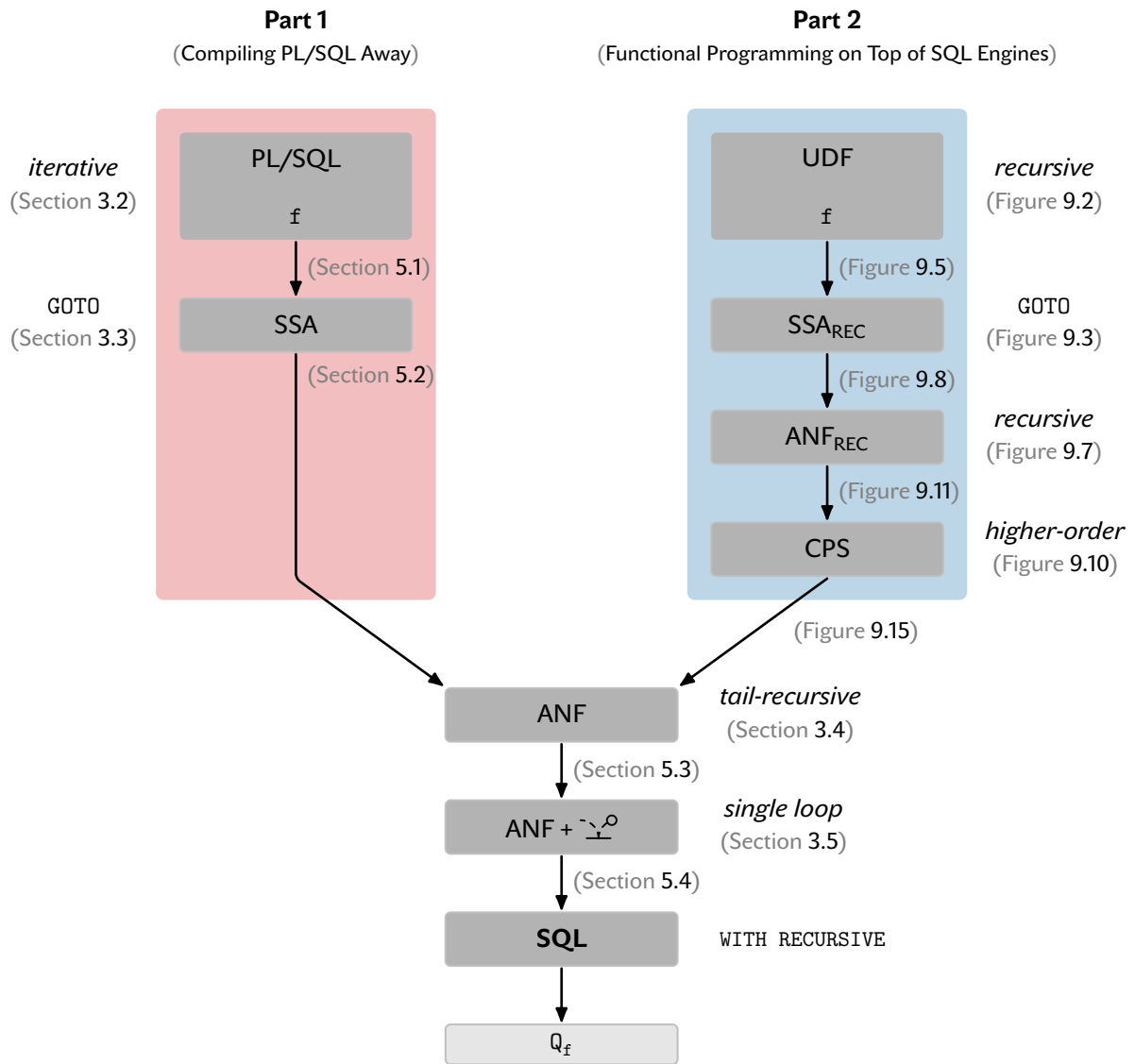


Figure 1.3.: Here is an overview of the two compilation pipelines described in Part 1 and Part 2. While the front ends are different, the back end is the same and can be reused.

COMPILING PL/SQL AWAY

Avoid PL/SQL if you can...



2.

“Move your computation close to the data” [22] is decades-old advice but still very relevant today. The database community has been aware for quite some time that it is not efficient to frequently move large amounts of data, *e.g.*, between a database server and its clients. A database client typically communicates with the server through (1) an inter-process communication system (IPC) provided by the operating system (this can be used when the server and client processes run on the same machine), or (2) over a network. Network communication, however, is typically much slower than using an IPC [23].

The overhead involved can become the most time-consuming part of the application. This is especially true if the application is implemented in a way that requires frequent communication with the database server, for example, if the application is implemented in an imperative programming language and uses SQL queries to retrieve data from the database server. In such cases, the application must send the SQL queries to the database server, wait for the server to execute the query, and then receive the result. This process is repeated for each query, which can be very inefficient. Therefore, it is generally recommended to move the computation closer to the data, instead of moving the data to the application. However, this is not always a trivial task, especially when the computation is complex, or implemented imperatively.

In this chapter, we will discuss the problem of moving computation closer to the data in more detail, and why PL/SQL should be avoided. At first we will have a look at the typical control flow of database-backed applications and discuss the associated overhead in Section 2.1. We will then discuss several problems of PL/SQL execution in Section 2.2. We will then proceed to discuss PostgreSQL’s implementation of PL/SQL. In Section 2.3 we will examine where the context switching overhead occurs. Finally, we will make this concrete in Section 2.4 and further explain why it is generally recommended to avoid PL/SQL if possible.

2.1. Context Switching

In many cases, database-backed applications are implemented using general-purpose programming languages. A typical pattern found in such applications is code that submits SQL queries to retrieve some data from the database system. This data is then processed iteratively in a row-by-row fashion [24]. The resulting control flow of such applications is sketched in Figure 2.1a. Each time the application issues a SQL query, there is some amount of overhead , *e.g.*, because of context switching, network latency, or IPC overhead [25]. Upon receiving, the database system executes the query and returns the result , which again is affected by overhead. Both directions

[22]: Rowe et al. (1987), ‘The PostgreSQL Data Model’

[23]: Bales (2015), *Beginning Oracle PL/SQL*

[24]: Gupta et al. (2020), ‘Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates’

[25]: Shao et al. (2020), ‘Database-Access Performance Antipatterns in Database-Backed Web Applications’

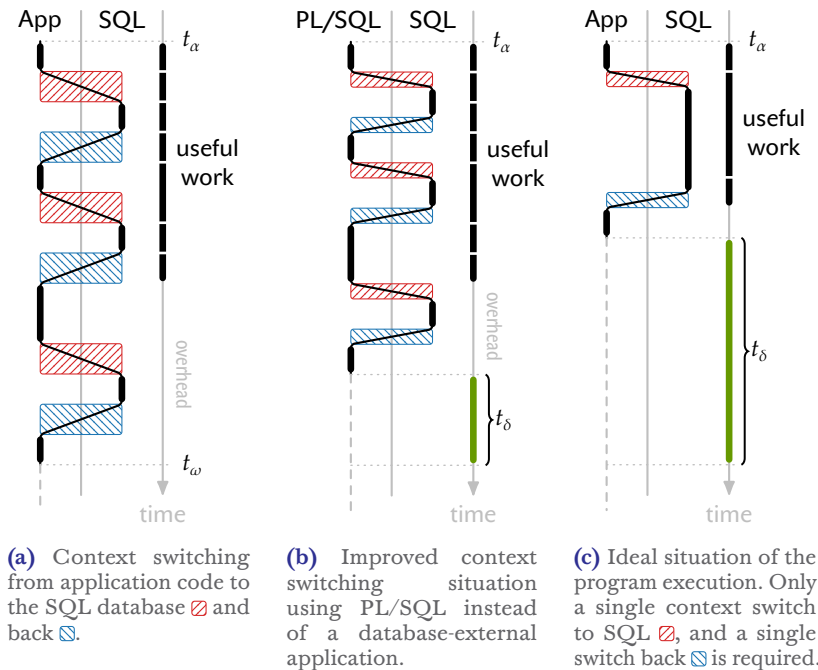


Figure 2.1.: Different execution patterns of database-backed applications.

take time in which no useful work is performed. This is generally undesirable. The situation is particularly dire when these queries are placed in tight loops. In that case, switching back and forth between the application and the database system occurs very frequently, which multiplies the overhead, and ultimately slows down the application.

PL/SQL and related solutions address part of this issue by providing an imperative in-database programming language close to the data the programs operate on. Instead of connecting to the database system remotely in order to retrieve some data, imperative PL/SQL programs that contain SQL queries can be sent to, and—in case of `STORED PROCEDURES`—be stored in the database directly. These programs are then executed by the database system, mitigating the need to switch back and forth between the application and the database system. Thereby implementing Rowe and Stonebraker’s advice. This is something we need. Because even though SQL has been around since the 1970s developers are still educated and thereby somewhat biased to prefer imperative code over declarative queries in SQL [26].

While that bias may not be an issue on its own, using imperative and declarative programming together creates a whole host of problems. Over the years, it has become common knowledge that PL/SQL is slow and should be avoided if possible [27]. Nevertheless, PL/SQL has been used for decades to implement complex database-backed applications [28]. As it continues to be used, the database community has taken on the challenge of making PL/SQL performance tolerable, resulting in several publications addressing the problem [24, 27, 29], [13–15].

PL/SQL’s slow execution can be attributed to the *structural impedance mismatch* happening between SQL’s declarative programming

[26]: Olston et al. (2008), ‘Pig Latin: A Not-so-Foreign Language for Data Processing’

[28]: Harris (2020), *A (Not So) Brief But (Very) Accurate History of PL/SQL*

[24]: Gupta et al. (2020), ‘Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates’

[27]: Ramachandra et al. (2018), ‘Froid: Optimization of Imperative Programs in a Relational Database’

[29]: Simhadri et al. (2014), ‘Decorrelation of user defined function invocations in queries’

[13]: Duta et al. (2020), ‘Compiling PL/SQL Away’

[14]: Hirn et al. (2020), ‘PL/SQL Without the PL’

[15]: Hirn et al. (2021), ‘One WITH RECURSIVE is Worth Many GOTOs’

paradigm, and PL/SQL’s imperative paradigm. In practice, this means that classical interpretation-based systems such as PostgreSQL require *two* evaluation contexts. While SQL is evaluated in a set-oriented, plan-based manner, PL/SQL entails statement-by-statement interpretation. Figure 2.1b shows this typical switching pattern between PL/SQL and SQL. As is the case for database external applications, PL/SQL encounters frequent context switching effort. For each embedded SQL query, the PL/SQL interpreter invokes the SQL executor \boxtimes , which takes time. The query is executed and the result is returned, again incurring some overhead \boxtimes . This design, too, can result in a lot of context switching overhead, which slows down execution. A key observation is that both, (1) database external applications in Figure 2.1a, and (2) database internal solutions using PL/SQL in Figure 2.1b show the exact same—*problematic*—execution pattern. Assuming that both programs perform the same computation, and that both the query runtimes and the useful work \blacksquare take about the same amount of time, PL/SQL is likely to be faster by time t_δ . This is because the context switching overhead is reduced compared to the database external solution. PL/SQL has managed to improve the situation somewhat, but the fundamental problem is still there and is still causing performance problems. Solving the performance issues requires solving the structural impedance mismatch, and therefore requires to completely rethink PL/SQL execution.

RDBMSs can be classified by a multitude of characteristics. One important criterion is the implementation of the execution strategy. The most common execution methods are (1) Volcano-iterator interpretation (PostgreSQL, SQLite3, MySQL), (2) vectorization (Snowflake, Vectorwise, DuckDB), and (3) data-centric code generation (Hyper, Umbra) [30].

In this work, we subscribe to the radical idea to eliminate PL/SQL altogether, and instead replace it with a computationally equivalent, probably recursive SQL query. After making this transition, there are no more context switches between the PL/SQL interpreter and the SQL executor. In fact, no PL/SQL interpreter is required at all. This situation is depicted in Figure 2.1c. There is some initial overhead \boxtimes when issuing the query, and some overhead when the result is returned \boxtimes . However, the actual computation requires no context switches at all and is completely executed by the SQL executor.

[30]: Kersten et al. (2018), ‘Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask’

Hyper and Umbra both have support for imperative in-database UDFs. The fundamental context switching problem with UDFs still exists in these systems, even though they compile UDFs and SQL into a single program representation.

2.2. Drawbacks of PL/SQL Evaluation

Besides context switching, PL/SQL performance can be bad for a variety of other reasons. While we describe the reasons in this thesis mainly in context of PostgreSQL, other systems show similar behavior. We will briefly discuss the drawbacks in this section. The following list is not exhaustive, but highlights the most important issues.

[31]: Moden (2005), *Calculating Work Days*

[32]: Fritchey et al. (2014), 'Row-by-Row Processing'

Row by agonizing row (RBAR) Execution. Database systems are designed to process sets of rows at a time. PL/SQL UDFs, however, force the system to invoke them for each input row individually. This mode of execution has been coined RBAR and is inefficient [31, 32]. Instead of using fast bag-oriented operations, the systems often have to resort to less efficient algorithms. Because the database system can take advantage of parallelism and bulk processing techniques, bag-oriented SQL operations are usually faster to process. For example, think of nested loop joins as opposed to hash joins. While nested loop joins have a complexity of $O(n \times m)$, hash joins use sophisticated data structures to reduce this complexity to $O(n + m)$. RBAR execution may prevent the use of such algorithms. This is because function calls can, *e.g.*, hide implicit joins.

Optimization. Query optimizers historically treat PL/SQL UDFs exclusively as black boxes, because the database system knows nothing about the function's behavior. This can negatively impact planning of queries containing calls to such functions, because the system's optimizer can not estimate the cost-factor. In case of table-valued functions (TVFs), the number of returned rows is also unknown, which can eventually result in suboptimal query plans.

[33]: *PostgreSQL 15 Documentation*

PostgreSQL version 12 added support for dynamic *function optimization information* annotations. A *planner support function* can provide the optimizer with additional knowledge about the cost-factor the function introduces, and the estimated number of returned rows. Previous to version 12 it was only possible to provide a static cost-factor and a static number of returned rows. A planner support function can be attached to a UDF during creation by specifying the `SUPPORT` clause. However, these functions must be written in C, which limits accessibility for users [33, §38.11]. Furthermore, the optimizer still treats the function as a black box and cannot optimize the execution of the function itself. It can only use the information provided to estimate the cost factor and the number of rows returned. This is a step in the right direction, but it does not solve the problem.

Interpretation. PL/pgSQL is implemented as an abstract syntax tree (AST) interpreter. The ASTs are created by the parser and are cached per database session. Caching is the only optimization applied, however. Current PL/pgSQL (shipped with PostgreSQL version 15) does not apply any defacto standard techniques such as constant propagation or folding, or dead code elimination.

2.3. Behind the Scenes of PL/SQL

PostgreSQL is an ideal platform to learn about *one* possible way how PL/SQL is implemented in a respected and well-known database system. An advantage of studying PL/SQL in PostgreSQL is the open-source nature of the database system, its robustness, and its reputation for adherence to industry standards. Even though the fine details will likely vary between different systems, the overall principle architectural issues of PL/SQL translate to other systems.

The execution of a PL/PGSQL function is handled by the PostgreSQL system-function `plpgsql_call_handler`¹. Execution proceeds in four basic steps:

1. A connection to the database using the server programming interface (SPI) is established. SPI is a C-level interface exposing the database internals. It can be used to run SQL queries inside language C UDFs. Several functions simplify access to the parser, planner, and executor.
2. PL/PGSQL functions are stored as strings in the database. Therefore the functions must be compiled before they can be executed using `plpgsql_compile`². The resulting data structures are cached until invalidation or until the end of the current database session. This prevents re-compilation on every function invocation which saves time.
3. PL/PGSQL's execution entry point is `plpgsql_exec_function`³. This function (1) prepares all required temporary data structures for execution, (2) calls the main interpreter function which returns the final result, (3) cleans up the temporary data structures, and (4) returns the final result to the caller.
4. Disconnect from the SPI manager, which will internally release the memory that is no longer needed.

All of these steps, with the exception of step 2, must be performed each time a PL/PGSQL function is called.

2.3.1. Embedded SQL

For each embedded SQL query, the system must switch from the PL/PGSQL interpreter to the SQL executor and back. PL/PGSQL does not implement its own expression evaluator, instead it completely relies on the SQL executor of the system. This functionality is handled by function `exec_eval_expr`⁴, which performs the following steps:

1. If the query is executed for the first time, plan the query and add it to the plan cache. Otherwise, retrieve the plan from the cache instead. Caching reduces the overhead associated with repeatedly parsing, planning, and optimizing queries.
2. Execute the query. PL/PGSQL differentiates between *simple expressions* and regular queries to optimize performance:
 - ▶ If the query is a simple expression, PL/PGSQL bypasses SPI and uses PostgreSQL's expression evaluator directly. A query qualifies as simple, if (1) it is a plain **SELECT** query

1: Source location:
`src/pl/plpgsql/src/pl_handler.c`

2: Source location:
`src/pl/plpgsql/src/pl_comp.c`

3: Source location:
`src/pl/plpgsql/src/pl_exec.c`

4: Source location:
`src/pl/plpgsql/src/pl_exec.c`

5: Source location:
src/pl/plpgsql/src/pl_exec.c

without any input tables, (2) there are no aggregates and no window-functions, and (3) the following clauses are not used `WHERE`, `HAVING`, `GROUP BY`, `ORDER BY`, `DISTINCT`, `LIMIT`, `OFFSET`. Function `exec_simple_check_plan`⁵, implements these checks. Bypassing SPI is extremely beneficial, because the runtime impact of creating a full executor instance is significant. Performance would be much worse without this optimization.

- ▶ If the query is not simple, a full executor instance must be created. Internally, the engine's functions `ExecutorStart`, `ExecutorRun`, and `ExecutorEnd` are used to create the executor instance (*i.e.*, copy the cached plan into a runtime data structure and instantiate the query's placeholders), run the query, and to free temporary memory contexts, respectively.

3. Check that the query returned exactly one column and at most a single row, since PL/PgSQL variables are single scalar values. Table-valued variables are not supported. Therefore, multiple return values cannot be handled in a meaningful way.
4. Return the result.

The PL/PgSQL query invocation is well optimized, but is still the main contributor to the context switching overhead $\boxtimes + \boxtimes$ during interpretation. PostgreSQL generally tries to use *plan caching* to ensure, that such embedded queries are compiled and optimized ideally only once on its first encounter during interpretation. While fetching plans from the cache still takes time, it is faster than having to plan and optimize the query completely.

PL/PgSQL reuses the `PREPARE`-statement infrastructure for the plan cache. Prepared statements can be executed with either a *generic plan*, or a *custom plan*. Generic plans are static for all executions, while custom plans are generated using the parameter values given in that call. Custom plans create more planning overhead, but can be more efficient to execute in some situations because the planner can use the actual parameter values, possibly resulting in a better plan [33, `$PREPARE`]. Since PostgreSQL version 12, it is possible to change how prepared statements are executed. This can be configured using the `plan_cache_mode`, which by default is set to `auto` [33, §20.7 Query Planning]. A suboptimal configuration of this parameter potentially introduces additional planning overhead \boxtimes , which can slow down PL/PgSQL interpretation.

Other database systems with PL/SQL interpreters show similar behavior, which ultimately led to the *Froid* [27, 34] and *Aggify* [24] efforts in Microsoft SQL Server [35], for example. These approaches also attempt to eliminate PL/SQL to optimize performance. We will discuss them in more detail in Sections 7.2 and 7.3.

[27]: Ramachandra et al. (2018), 'Froid: Optimization of Imperative Programs in a Relational Database'

[34]: Ramachandra et al. (2019), 'BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid'

[24]: Gupta et al. (2020), 'Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates'

[35]: Microsoft SQL Server 2022 Documentation

2.4. Case Study: UDF route

We will now investigate the performance of PL/pgSQL UDFs using the `route` UDF as an example. UDF `route` of Figure 2.2 is implemented in PostgreSQL’s PL/SQL dialect PL/pgSQL [36]. A call `route(s, d, ttl)` returns an array of all intermediate nodes, on the path between a source node s and destination d , in a point-to-point network (see Figure 2.4a as an example). Each row (h, t, v, c) in table `connections` (Figure 2.4b) indicates that the cheapest path from h to t goes via hop v for an overall cost of c . Function `route` returns `NULL`, should the costs exceed the budget ttl . A selection of the resulting paths computed by `route` can be seen in Figure 2.4c.

[36]: PostgreSQL 15 PL/pgSQL Documentation

PL/pgSQL UDF `route` contains multiple SQL expressions like `loc <> dest.ttl`, `ttl < hop.cost`, or `route || loc` (see lines 10, 14, and 18 in Figure 2.2). These SQL expressions qualify as simple and are therefore evaluated without SPI. Instead, the *fast path* directly uses the system’s SQL expression evaluator.

However, `route` also contains the SQL `SELECT`-block Q_1 which queries the routing table for the next hop from `loc` towards `dest`. Q_1 has two free variables `loc` and `dest` (see Line 13 of Figure 2.2) which have to be instantiated for each invocation. As explained in Section 2.3.1, *each of the potentially many evaluations* of the embedded query requires:

1. the instantiation of the runtime data structures for Q_1 ’s plan,
2. creating an instance of the SQL executor to evaluate Q_1 , and
3. releasing and cleaning up of temporary data structures, before returning the result back to the PL/pgSQL interpreter.

The overhead of such embedded queries typically constitutes the lion share of the friction between SQL and PL/SQL. A call to UDFs like `route` is typically embedded in a top-level SQL query, say Q_0 . Here, the query finds paths that consists of more than two hops:

```
SELECT c.here, c.there, route(c.here, c.there, 10)
FROM   connections AS c
WHERE  cardinality(route(c.here, c.there, 10)) > 2;
(Q0)
```

Query Q_0 executes UDF `route` for each row in table `connections`, which results in context switches in two directions:

Q \rightarrow f [switch from the top-level query Q to UDF f] For each row of table `connections`, the `SELECT` and `WHERE` clauses in Q_0 invoke the PL/SQL interpreter to evaluate the embedded UDF `route`.

f \rightarrow Q [switch from UDF f to embedded query Q] UDF `route` contains the assignment of variable `hop` in Line 11 of Figure 2.2. Each execution calls on the SQL engine to evaluate the embedded query Q_1 , which required plan instantiation and cleanup, as described in Section 2.3.1. Note, that this assignment is located inside a `WHILE` loop and will therefore be iterated. One invocation of `route` can lead to several `route \rightarrow Q1` context switches.

The resulting switching between the PL/SQL interpreter and SQL executor is depicted in Figure 2.3. From top to bottom, execution

```

1 CREATE FUNCTION route(source node, dest node, ttl int)
2 RETURNS node [] AS $$
3 DECLARE
4     route node[];           -- path between source and dest
5     loc node;              -- current location in network
6     hop connection;       -- next hop leading from loc to dest
7 BEGIN
8     loc := source;        -- move to source
9     route := array[loc];  -- path starts at source
10    WHILE loc <> dest LOOP  -- while dest has not been reached:
11        hop := (SELECT c   Q1[.,.] -- consult routing table
12              FROM   connections AS c -- to find next hop
13              WHERE  c.here = loc AND c.there = dest);
14        IF ttl < hop.cost THEN -- bail out if
15            RETURN NULL;      -- budget exceeded
16        END IF;
17        loc := hop.via;      -- move to found hop
18        route := route || loc; -- add hop to path
19    END LOOP;
20    RETURN route;          -- return constructed path
21 END;
22 $$ LANGUAGE PLPGSQL;

```

Figure 2.2.: Original PL/SQL UDF `route`. $Q_1[.,.]$ denotes an embedded SQL query containing the free variables `loc` and `dest`.

starts in the SQL context of top-level query Q_0 . Every invocation of UDF `route` requires a context switch $Q \rightarrow f$ to the PL/SQL interpreter. Each time embedded query Q_1 needs to be evaluated, interpretation of UDF `route` is suspended to establish a new SQL context. This, again results in a context switch \boxtimes to create the executor instance, and another context switch back \boxtimes when finished. As this query is placed inside a `WHILE`-loop (marked \blacksquare), this may happen many times. The execution of `RETURN` in Line 20 of `route` marks the end of PL/SQL interpretation, and control returns $f \rightarrow Q$ to the top-level context of Q_0 until the next invocation of `route` occurs.

These context switching efforts add up and significantly contribute to the overall execution time: in the time frame $t_\omega - t_\alpha$, only the time slices \blacksquare perform useful work such as plan evaluation or statement interpretation. The rest of the time is spent on context switching.

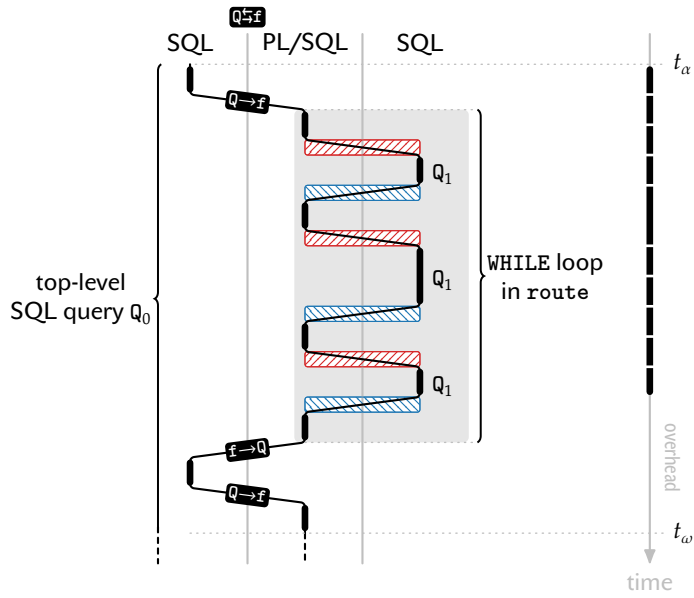


Figure 2.3.: Context switching as top-level query Q_0 executes.

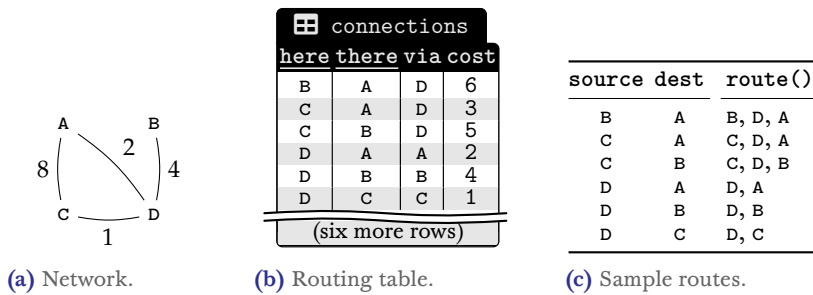


Figure 2.4.: Sample network with distance-vector routing table.

One Way to Trade PL/SQL for SQL 3.

Without drastic measures, PL/SQL's context switching issue cannot be fixed. Correcting this requires eliminating the aforementioned structural impedance mismatch, which is the root cause of the problem. So our goal here is to remove PL/SQL completely. The following sections describe a compilation method that is capable of compiling iterative PL/SQL UDFs (with possibly deeply nested control flow) into recursive yet plain SQL queries.

The compiler is structured as a series of *stages*. The first stage receives an imperative PL/SQL UDF f , and the last one emits a plain declarative SQL query Q_f . The final query Q_f no longer requires a PL/SQL interpreter anymore because the entire computation is expressed in SQL. This eliminates the structural impedance mismatch. The compiler is not specific to any database system, as it is realized as a source-to-source translation. The underlying RDBMS remains unchanged. This means, that the resulting query Q_f can be executed on any RDBMS with support for `WITH RECURSIVE`, and `LATERAL` joins. Both SQL language features are widely supported and have been around for more than two decades [37, 38]. This work thus also provides a foundation upon which PL/SQL support can be built for systems that do not have a PL/SQL interpreter at all.

[37]: Eisenberg et al. (1999), 'SQL:1999, Formerly Known as SQL3'

[38]: SQL:1999 Standard. Database Languages—SQL—Part 2: Foundation

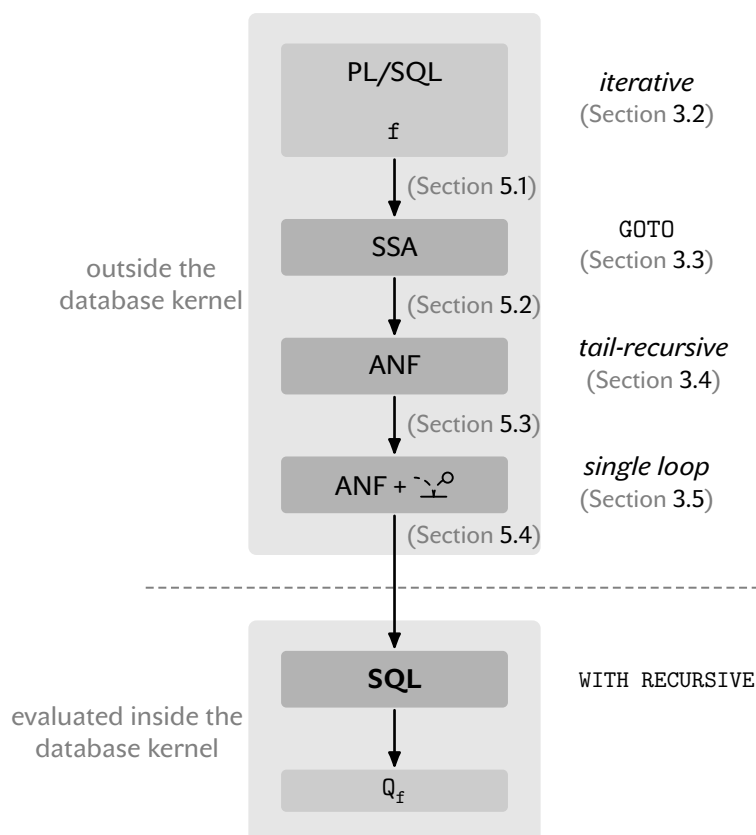


Figure 3.1: Compilation stages and intermediate UDF forms.

3.1. The Expressive Power of SQL

Before proceeding with the description of the compilation chain, it is important to understand the required subset of SQL and its expressiveness, since this is the target language. SQL is optimized for expressing database queries in a declarative manner, rather than being a general-purpose programming language. Loops, statement sequencing, and variable assignment do not have direct language support. However, these are the core features of PL/SQL, so an appropriate mapping to SQL is required. SQL provides developers with a great deal of useful functionality that continues to grow with each revision of the SQL standard. The SQL:1999 standard is particularly relevant to this work, as this version introduced **LATERAL** joins and *recursive common table expressions (CTEs)*—which made SQL a Turing-complete language.

Although SQL is a standardized language, virtually all relational database systems deviate at least slightly from the standard and have developed their own dialect with syntactic and semantic differences. In practice, this means that it is not simple to take an SQL query from one system and run it on another¹. To simplify the discussion, we will argue mainly using the SQL dialect of PostgreSQL, as this problem is not specific to this work, but complicates migration from one system to another in general. Furthermore, PostgreSQL tries to be as standards-compliant as possible.

1: This is known as *vendor lock-in*, and makes a customer dependent on a particular product because switching to another product involves significant switching costs.

[33]: PostgreSQL 15 Documentation

Non-recursive CTEs. A CTE provides a way to structure and simplify (complex) queries by binding an intermediate relation to a name. The relation exists only within the scope of the statement and can be used multiple times. CTEs are related to temporary tables, but without the overhead involved with creating an actual table [33, §7.8 WITH]. The example in Figure 3.2 defines the CTEs cte_1 , cte_2 , ..., cte_k . Without using the **RECURSIVE** keyword, cte_1 can be used in $Q_{2..k}$ as well as in Q_f , but not in its defining query Q_1 . Similarly, cte_2 can be used in $Q_{3..k}$ and in Q_f , but not in $Q_{1,2}$. The bindings are performed in a sequential manner, and any form of self-reference is prohibited. In a sense, non-recursive CTEs can be seen as a simple syntactic convenience.

```

1 WITH
2   cte1(a1,...,an) AS (Q1),
3   cte2(b1,...,bm) AS (Q2),
4   :
5   ctek(k1,...,ko) AS (Qk)
6 Qf

```

Figure 3.2: This query defines the non-recursive CTEs $cte_{1,2..k}$. Previously defined CTEs 1..n-1 can be used in Q_n , which defines cte_n .

Recursive CTEs. Using the **RECURSIVE** keyword is a completely different story, and allows for a much more powerful language feature. The previous restriction that the definition of cte_n , Q_n , cannot use itself recursively does not apply. Figure 3.4a shows a recursive CTE with the name T. The definition of a recursive CTE T is divided into two parts connected by a **UNION [ALL]** operation. Note that this is a *recursive UNION [ALL]* operation and should not be confused with the regular set operation. The first operand of the recursive **UNION [ALL]** always contains the initialization query q_1 , the iterated query q_m (read: “q loop”) is the second operand. While q_m can refer to T (with certain restrictions), the initialization query q_1 must not refer to T. Although the **UNION [ALL]** keyword has been reused to express recursive CTEs, we are dealing with a different semantics in this

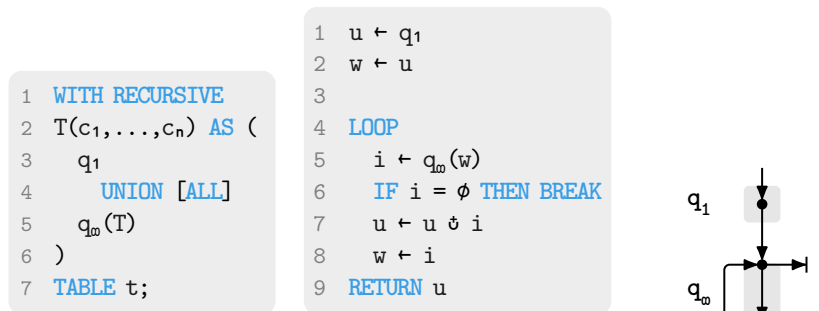
context. While recursive CTEs are typically used to express computations over hierarchical structures [39], their *fixpoint semantics* generally allow any Turing-complete computation to be expressed. PL/SQL is also Turing-complete, so this is critical.

Figure 3.3 shows, for a small sample of database systems, when recursive CTEs were introduced. While this feature is widely available today, it took decades for most SQL dialects to become Turing complete. Prior to this, Turing-complete computations could not be expressed in pure SQL. This is one of the reasons why PL/SQL was invented.

Fixpoint-based Semantics. Fixpoint theory [40] is a way to formally describe the semantics of recursive programs and is used to specify the semantics of recursive CTEs. Essentially, recursive CTEs compute the least *fixpoint* $T = q_1 \text{ UNION [ALL] } q_m(T)$. This fixpoint is guaranteed to exist, and to be unique, when q_m is *monotonic* [41, 42]. Monotonicity means that adding rows to the input set will not remove or change any rows in the result. This leads to *syntactic restrictions* on q_m that prevent any use of negation (e.g., `NOT EXISTS`), `INTERSECT/EXCEPT`, outer joins, deduplication of rows via `DISTINCT`, or grouping and aggregation. However, it enables the *semi-naive* evaluation of q_m over only the rows produced in the *immediately preceding iteration* [43]. Recursive CTEs initially evaluate the non-recursive SQL query q_1 once, then q_m is iterated. Figure 3.4b shows this loop-based semi-naive evaluation algorithm used by most DBMSs. This simple linear computation scheme corresponds to the call graph of Figure 3.4c. Three table-valued variables are required:

- w (working table):** contains the rows produced by the immediately preceding iteration. This table can be accessed by q_m using the table name T .
- i (intermediate table):** holds the rows of the current evaluation of q_m . The recursive CTE exits the LOOP, if i is empty (see Lines 5 and 6 in Figure 3.4b).
- u (union table):** collects the rows returned by q_1 and any intermediate tables computed by q_m . This table defines the result of the CTE.

The monotonicity constraint on q_m ensures that the loop in Figure 3.4b conforms to the original SQL:1999 fixpoint semantics.



(a) SQL syntax for recursive CTEs. (b) Semi-naive operational semantics for recursive CTEs. (c) Recursion in a CTE.

[39]: Shalygina et al. (2017), ‘Implementing Common Table Expressions for MariaDB’

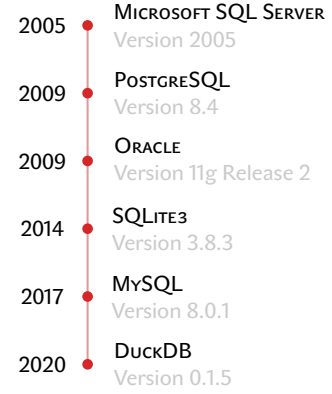


Figure 3.3.: This timeline shows a selection of database systems and their respective year and version when recursive CTEs were added.

[40]: Tarski (1955), ‘A lattice-theoretical fixpoint theorem and its applications.’

[41]: Bancilhon et al. (1986), ‘An Amateur’s Introduction to Recursive Query Processing Strategies’

[42]: Finkelstein et al. (1996), *Expressive Recursive Queries in SQL*

[43]: Bancilhon (1986), ‘Naive Evaluation of Recursively Defined Relations’

Figure 3.4.: SQL syntax and imperative operational semantics for recursive SQL CTEs.

3.2. The PL/SQL Language

The following sections describe a formal way to transform any (non-recursive) imperative program into a program representation that can then be expressed with a single (recursive) SQL query.

The PL/SQL language is composed of *imperative* language constructs, and of *embedded* SQL expressions and queries. This design integrates seamlessly with SQL while allowing imperative computation to be expressed in the database. This is supported by the fact, that both languages use SQL's type system. Using the same data types removes the need to convert query results into an appropriate data type in PL/SQL, which saves time. Because of the nesting of SQL and PL/SQL, we are effectively dealing with *two* programming languages at once. In such a situation, the programming language community typically refers to PL/SQL as the metalanguage and SQL as the object-language. We will use this terminology as well.

The key elements of PL/SQL are

- ▶ variable assignments $v := (a)$,
- ▶ control flow in terms of, e.g., **IF ... ELSE**, **LOOP**, **WHILE**, **FOR**, **EXIT**, **CONTINUE**, and
- ▶ embedded SQL queries and expressions.

Any scalar, array, or composite data type can be used as PL/SQL function arguments, return type, and variables. There are two kinds of PL/SQL functions: (1) Scalar functions return a single value of type τ , or **NULL**. This is done via the **RETURN** statement. The return value of a function cannot be left undefined. If control reaches the end of the function without passing through a **RETURN** statement, PL/SQL throws a run-time error. (2) TVFs return a *table* of values, or no values at all. The individual rows to return are specified using the **RETURN NEXT** or **RETURN QUERY** statements. A final **RETURN** statement without arguments can optionally be used to indicate that the execution is finished [36].

[36]: *PostgreSQL 15 PL/pgSQL Documentation*

f	::=	CREATE FUNCTION $v(\overline{v} \ \tau)$ RETURNS [SETOF] τ AS \$\$ p \$\$ LANGUAGE PLPGSQL STABLE ;	PL/SQL UDF
p	::=	DECLARE d ; BEGIN s ; END	UDF body
d	::=	$v \ \tau \mid v \ \tau := a \mid d; d$	variable declarations
s	::=	$v := a$ IF a THEN s ; [ELSE s ;] END IF LOOP s ; END LOOP WHILE a LOOP s ; END LOOP FOR v IN $a..a$ LOOP s ; END LOOP EXIT CONTINUE RETURN a RETURN NEXT a RETURN QUERY a $s; s$	statements table-valued statements
a	::=	SQL query [\overline{v}]	boxed SQL query
v	::=	\langle identifier \rangle	variable/function name
τ	::=	\langle scalar SQL type \rangle	scalar value type

Figure 3.5.: PL/SQL language subset covered by the compiler.

Figure 3.5 formally defines the admissible PL/SQL dialect that the compiler can handle. The subset of PL/SQL does not restrict the expressible control flow, and allows various forms of loops which can be nested, cut short via `CONTINUE`, or left early using `EXIT` or `RETURN`. The source of context switching between the SQL executor and the PL/SQL interpreter are embedded SQL queries, expressed in terms of non-terminal *a*.

Blackboxing. Internal details of embedded queries and expressions are not required during the following translation steps. The non-terminal *a* is therefore a parameterized black box `Q`. It is only at the very last compilation step that the parameters are substituted into the actual SQL code.

Unsupported PL/SQL Features. PL/SQL has a few features that are not supported by the compilation technique described in the following Sections 3.3 to 3.6.

Dynamic SQL. PL/SQL allows users to build SQL statements at the time of execution. Since the final compilation target is a single pure SQL query, this feature is not compilable, as it would require modifying the source code of the current SQL query at runtime. This is not possible in SQL.

Exception Handling. PL/SQL allows users to define custom exception handlers. This can be used to catch and handle errors that occur during execution, *e.g.*, when a division by zero is performed, or when the connection to the database is lost [33, Appendix A. PostgreSQL Error Codes]. It is not possible to compile such handlers, because SQL does not support exception handling.

INSERT/UPDATE/DELETE Statements. With PL/SQL, it is possible to modify the state of the database by inserting, updating, or deleting rows in tables. PL/SQL supports this because each statement is executed in its own SQL execution context. After compilation, there is only one SQL execution context. Nesting of `SELECT` statements with `INSERT`, `UPDATE`, and `DELETE` statements to this extent is not supported in SQL. In particular, it is impossible to modify a table and read its modified state in the same query, as this would violate the ACID property of the system [44, 45].

Data definition language (DDL) Statements. Statements such as `CREATE`, `DROP`, and `ALTER` can be executed by PL/SQL, but they cannot be placed within a regular `SELECT` query. Therefore, these statements cannot be compiled either.

Cursor Loops. These loops are used to iterate through the result of a query. While it is not technically impossible to compile these loops, the compiled version is usually inefficient. This is because cursors are not a language feature of SQL. This means that it is not possible to read the result of a query row by row using a recursive CTE without re-executing the query, or materialization of the entire query result first.

[33]: PostgreSQL 15 Documentation

[44]: Gray (1981), ‘The Transaction Concept: Virtues and Limitations (Invited Paper)’

[45]: Haerder et al. (1983), ‘Principles of Transaction-Oriented Database Recovery’

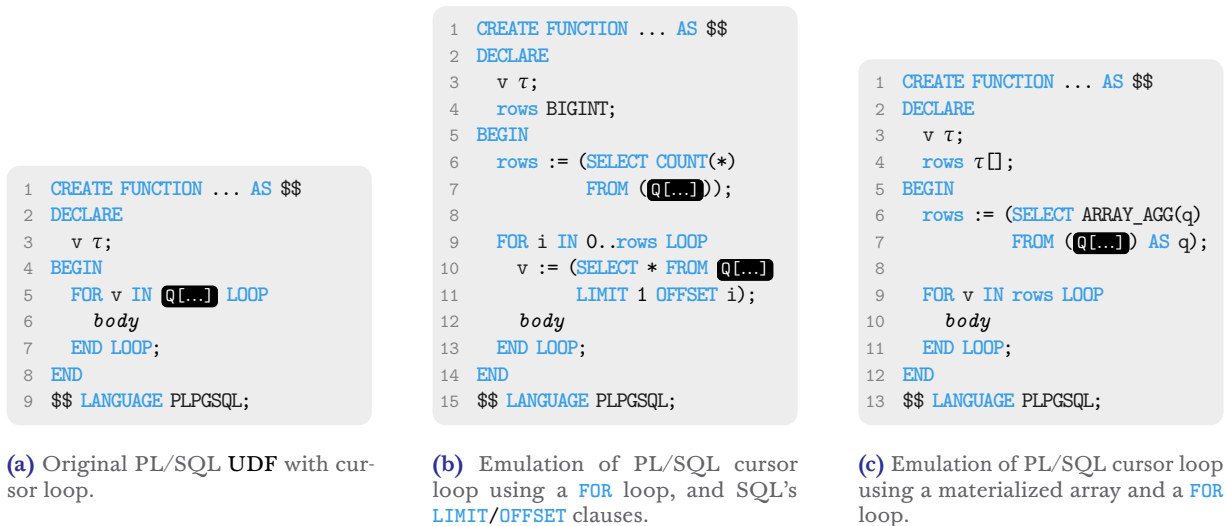


Figure 3.6.: Original PL/SQL cursor loop and two emulation techniques.

However, cursor loops can be mimicked with the help of supported control flow elements. Figure 3.6a defines a PL/SQL function that uses a cursor loop to iterate over the result of query `Q[...]`. Figures 3.6b and 3.6c depict feasible emulation techniques for cursor loops that can be used. Both options have drawbacks. The technique in Figure 3.6b has to evaluate query `Q[...]` $n + 1$ times. Once to determine the number of rows returned by the query, and once per iteration. It is important to note that the query results must be ordered to ensure that we iterate over the entire result set in the correct order. Without ordering, there is no guarantee that this will happen. The entire query result is materialized as an array in the alternative technique shown in Figure 3.6c. If the query returns many rows, the array contains many entries, which can be problematic after compilation. Since our compilation target involves a recursive CTE, the array needs to be copied during each iteration of the CTE, resulting in excessive overhead and potentially a slowdown in execution time.

3.3. PL/SQL Control Flow in Terms of GOTO

In general, programming languages come with a lot of features that make them easy to use. This includes various types of loops, conditional branching, *etc.* However, these features are not technically required and make the compilation process more difficult. Therefore, many compilers use small and simple core IRs, that are easy to reason about and easy to compile to. This compiler follows the same design philosophy. The first compilation pass reduces language complexity and transforms the body of the input UDF into a simple imperative control flow graph (CFG). All LOOP-statements and all control flow is expressed exclusively in the form of IF ... ELSE ... and GOTO-statements. Labeled basic blocks with ;-separated statement sequences are used to organize the code. All blocks end with either GOTO κ (passing control to the basic block with label κ), or RETURN. The syntax of this IR is defined in Figure 3.7.

A CFG [46] is a directed graph $G = (V, E)$ program representation with nodes V and edges E . Each CFG has a unique entry point `start`, where all control flow enters. In this work, all CFG nodes represent a *basic block* containing a sequence of statements. The edges indicate any jump or branching statements. Despite the simplicity of such a GOTO-based form, arbitrarily complex control flow patterns can be expressed. This simplicity is the key to streamlining the entire compilation. The construction of a CFG is typically based on the input AST and can be done very efficiently [47]. The translation from PL/SQL to the GOTO-based form as mapping \mathfrak{g} is described in detail in Section 5.1. Figure 3.8 visualizes the CFG for the PL/SQL UDF `route` of Figure 2.2. During the compilation, all embedded SQL expressions and queries are preserved. The non-terminal a in the PL/SQL grammar of Figure 3.5 and the GOTO IR in Figure 3.7 are identical. Loops in the input UDF materialize as cycles in the CFG. For example, the cycle `while` \rightarrow `loop` \rightarrow `meet` represents the WHILE-loop in UDF `route`. Nodes with multiple predecessors (*e.g.*, `while`) are called *join nodes*.

CFGs are a simple but very powerful tool for implementing control flow optimizations like inlining. Block inlining [48–51] can reduce

f	::= fun $v(\overline{v} \tau) : [\text{SETOF}] \tau \{\overline{b}\}$	top-level function
b	::= $\kappa : p s;$	labelled block
p	::= $v: \tau \leftarrow \phi(\kappa: \overline{v}); p$	SSA phi function
	ϵ	
s	::= $v \leftarrow a$	statement
	IF v THEN t ELSE t	
	t	block terminal
	EMIT a	table-valued result
	$s; s$	statement sequence
t	::= GOTO κ	jump to block κ
	RETURN a	return a value
a	::= SQL query $[\overline{v}]$	boxed SQL query
v	::= $\langle \text{identifier} \rangle$	variable/procedure name
τ	::= $\langle \text{scalar SQL type} \rangle$	parameter/return type
κ	::= $\langle \text{block label} \rangle$	jump target for GOTO

[46]: Allen (1970), ‘Control flow analysis’

[47]: Stanier et al. (2013), ‘Intermediate Representations in Imperative Compilers: A Survey’

[48]: Peyton Jones et al. (2002), ‘Secrets of the Glasgow Haskell Compiler inliner’

[49]: Chang et al. (1989), ‘Inline Function Expansion for Compiling C Programs’

[50]: Ferrante et al. (1987), ‘The Program Dependence Graph and Its Use in Optimization’

[51]: Waddell et al. (1997), ‘Fast and Effective Procedure Inlining’

Figure 3.7.: GOTO-based imperative intermediate form.

the number of basic blocks, and thereby the number of `GOTO` statements in a program. A basic block can be inlined into another block by simply replacing the `GOTO` κ statement with the statements of block κ . However, inlining can lead to exponential code growth, if κ has more than one predecessor. Therefore, inlining must be used carefully. In this work, we use inlining to simplify the CFG and to reduce the number of `GOTO` statements.

[52]: Cytron et al. (1991), ‘Efficiently Computing Static Single Assignment Form and the Control Dependence Graph’

2: Note: This is a property of the program code. During execution of a loop, for example, the same variable can be assigned several times.

[53]: Rastello et al. (2022), *SSA-based Compiler Design*

[5]: Novillo (2003), ‘Tree SSA a new optimization infrastructure for GCC’

[54]: Braun et al. (2013), ‘Simple and Efficient Construction of Static Single Assignment Form’

[55]: Choi et al. (1991), ‘Automatic Construction of Sparse Data Flow Evaluation Graphs’

[56]: Briggs et al. (1998), ‘Practical Improvements to the Construction and Destruction of Static Single Assignment Form’

[57]: Appel (1998), ‘SSA is Functional Programming’

Static single assignment form (SSA). Next, we transform the CFG into SSA [52]. In SSA, each variable is assigned *exactly once* in the program code². Once this property is established, SSA ensures *referential transparency*, just like programs implemented using pure functional programming languages do [53]. SSA simplifies data flow optimizations by making *def-use* chains explicit. SSA’s *phi functions* [52] are used to specify which variable versions are live for blocks with multiple predecessors (see block `while` in Figure 3.8, for example). This enables further well-known simplifications such as dead code elimination, unused variable detection, common subexpression elimination, or constant folding [5].

We will not go into the fine details of SSA construction, as it is well studied and described, *e.g.*, in [53]. However, we will give a brief overview because we have some unusual requirements that simplify the following compilation steps. The construction of the SSA property is a multi-step process. The two main steps are (1) phi function placement, and (2) variable renaming. Compilers usually try to insert the minimal number of phi functions to obtain so-called *minimal SSA*, *pruned SSA*, or *semi-pruned SSA* [54].

Pruned SSA: A program is in pruned SSA form if every variable v bound by a ϕ function $v \leftarrow \phi(\dots)$ is used by a non- ϕ expression [55].

Semi-pruned SSA: This form reduces the number of ϕ functions using the set of *block-local* variables (see `p0` in block `while` of Figure 3.8 for an example). Such variables never need a ϕ function [56].

Minimal SSA: Minimal SSA places ϕ functions only in join nodes, and only in blocks where different definitions of a variable v meet [52, 54] (see `loc1` in Figure 3.8).

Minimal SSA is advantageous for certain types of optimizations and is produced by most compilers. However, this work benefits from what Appel calls “*the crude approach*” [57]:

Crude SSA: Crude SSA places ϕ functions *for all live variables, in every node*. Compared to minimal SSA, this results in many more ϕ functions (see `dest1,2,3`, `ttl1,2,3`, `loc2,3`, `route2,3`, and `hop2,4` in Figure 3.8).

While placing so many (allegedly unnecessary) ϕ functions is a bad idea for most compilers, it is almost trivial to compute and makes the next compilation steps much easier. We will get back to this in the next section.

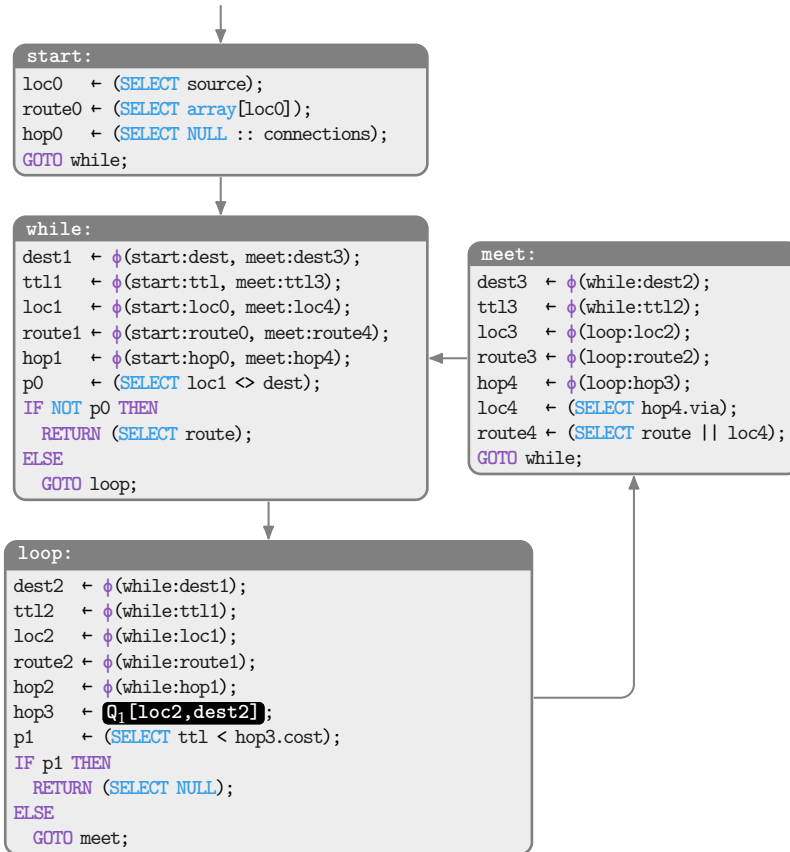


Figure 3.8.: CFG for UDF route with code blocks in SSA form.

3.4. Tail Recursion Replaces GOTO

SSA is typically intended to be used with languages that have scoping and a mutable state. However, SQL has no direct support for a mutable state. So we have to emulate it. This situation is similar to that of *pure* functional programming languages like HASKELL, for example. These languages also have no state, but that does not limit their expressiveness in any way. There are solutions to this problem on the functional programming side of the fence.

Consider the imperative code in Figure 3.9a. This program computes the COLLATZ conjecture³ [58] for input x and returns the number of steps n the algorithm took to terminate. Both variables are updated during execution (see Lines 4, 7 and 9 of Figure 3.9a). This state update is a side effect. We rely on the execution context to manage the state of these variables and assume the appropriate semantics when they are updated.

In contrast, the pure functional code in Figure 3.9b performs the exact same computation without the need for mutable variables. Instead, the state is passed explicitly as parameters to the function (see Lines 7, 8 and 11 of Figure 3.9b). Rather than relying on the execution context to manage the state of variables x and n for us, we use function arguments to emulate the state. This is something we can easily express using SQL. The elegance of this approach is that all data flow is explicit and no side effects are required. The state monad is a popular metaphor for this idiom in the functional programming community [59]. The COLLATZ conjecture is a simple

3: COLLATZ is defined as:

$$f(x) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod{2} \\ 3n + 1 & \text{if } n \equiv 1 \pmod{2} \end{cases}$$

Based on $f(x)$ we can define a sequence starting from any positive integer $x \in \mathbb{N}$:

$$a_i = \begin{cases} n & \text{for } i = 0 \\ f(a_{i-1}) & \text{for } i > 0 \end{cases}$$

The COLLATZ conjecture is that a_i ends in $\dots, 4, 2, 1$ for all $n \in \mathbb{N}$.

[58]: Lagarias (1985), ‘The $3x + 1$ Problem and its Generalizations’

[59]: Wadler (1993), ‘Monads for functional programming’

```

1 fun collatz(x int) : int {
2   n ← 0;
3   WHILE x != 1 LOOP
4     n ← n + 1;
5     IF x % 2 == 0
6     THEN
7       x ← x / 2;
8     ELSE
9       x ← x * 3 + 1;
10    END LOOP;
11    RETURN n;
12 }

1 fun collatz(x int) : int {
2   while(x int, n int) : int {
3     LET n1 = n + 1 IN
4     IF x != 1
5     THEN
6       IF x % 2 == 0
7       THEN while(x / 2, n1)
8       ELSE while(x * 3 + 1, n1)
9     ELSE n
10    }
11   while(x, 0)
12 }

```

Figure 3.9.: Imperative and purely functional computation of the COL-LATZ conjecture.

(a) Imperative program with mutable variables x and n .

(b) Pure functional program that emulates state using function arguments.

example, but this technique can be used to express arbitrarily complex state machines. This is the key to emulating mutable variables in SQL.

The compilation from the imperative code in Figure 3.9a to the pure functional code in Figure 3.9b is well-known and systematic. This is due to the correspondence between SSA and functional code (either in administrative normal form (ANF) or CPS form) that Kelsey [60] and Appel [57, 61] identified. This correspondence allows bi-directional compilation from SSA to functional code and vice versa, and has been the subject of extensive study in the programming language community. We are aiming for the *direct style* ANF IR [62]. It consists of conditionals, LET bindings, and function calls. Direct style means that there are no higher-order functions. This is important because SQL has no direct means of expressing such functions⁴.

The basic idea of how to compile SSA to ANF is relatively straightforward. In short, each basic block labeled κ becomes a function $\kappa()$. The sequence of assignment statements $v \leftarrow a$ in the block is expressed in terms of LET $v = a$ bindings. GOTO κ statements result in a *tail call* to function $\kappa()$. Figure 3.10 shows the grammar definition for our ANF language. Note that the non-terminals a , v , and τ are identical to the ones in the SSA grammar in Figure 3.7. This means that we can reuse the concrete values when compiling from SSA to ANF. To perform this step, we follow the work of Chakravarty, Keller, and Zadarnowski [9] who formalized the compilation of SSA

[60]: Kelsey (1995), ‘A Correspondence between Continuation Passing Style and Static Single Assignment Form’

[57]: Appel (1998), ‘SSA is Functional Programming’

[61]: Appel (1997), *Modern Compiler Implementation in ML: Basic Techniques*

[62]: Flanagan et al. (1993), ‘The Essence of Compiling with Continuations’

4: We will get back to this in Part “Functional Programming on Top of SQL Engines”, when we compile recursive UDFs into SQL.

[9]: Chakravarty et al. (2004), ‘A functional perspective on SSA optimisation algorithms’

Figure 3.10.: Intermediate functional language in ANF.

```

p ::= fun v( $\overline{v \tau}$ ) : [SETOF]  $\tau$  { $\overline{f}$ }           top-level function
f ::= v( $\overline{v \tau}$ ) :  $\tau = e$                        function definition
e ::= a                                           expression
    | v( $\overline{a}$ )
    | IF v THEN e ELSE e
    | LET v = a IN e
    | EMIT a; e                                     emit value
a ::= SQL query [ $\overline{v}$ ]                             boxed SQL query
v ::= <identifier>                               variable/function name
 $\tau$  ::= <scalar SQL type>                         parameter/return type

```

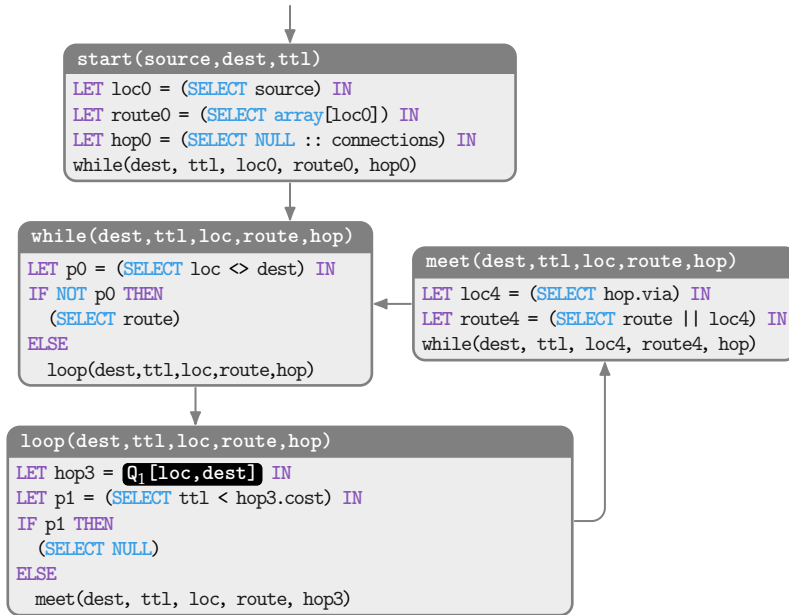


Figure 3.11. Call graph for UDF route after ANF conversion.

programs into ANF. We will instantiate our own version of this compilation in Section 5.2.

When we apply this idea to the `route` UDF in Figure 3.8, we end up with a set of mutually recursive functions `while`, `loop`, and `meet`. Each function takes arguments `(dest, ttl, loc, route, hop)`. Figure 3.11 shows the resulting function *call graph*. Note that the functions are all *top-level* and have the same arguments. The use of crude SSA yields this property. If we had used minimal SSA, the function arguments would be different for each of the functions. To ensure proper scoping for each identifier, the functions would then need to be nested.

ANF has a solid foundation of well-established optimizations [62, 63]. In our case, function *inlining* is the most important optimization, because it reduces the number of functions in the call graph, which is critical for the performance of the final SQL query. Figure 3.12 shows the final optimized version of UDF `route` in ANF. Functions `loop()` and `meet()` have been inlined into `while()`. This gives us a family with two functions. Inlining in ANF is generally applicable to all nodes with one predecessor. Function `start()` must not call itself, however. In general, if the input CFG contains n cycles, exhaustive inlining will result in a call graph of $n + 1$ functions (+1 for the `start()` function).

[62]: Flanagan et al. (1993), ‘The Essence of Compiling with Continuations’

[63]: Jagannathan et al. (1996), ‘Flow-directed inlining’

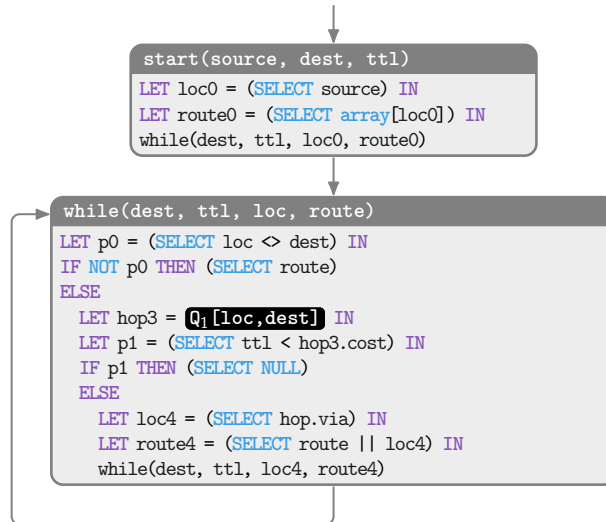


Figure 3.12.: Call graph for UDF route after inlining optimization.

3.5. Trampoline Style Tames Mutual Recursion

Complex UDFs that express (potentially deeply nested) looping computation and the simple iterative semantics of SQL’s `WITH RECURSIVE` appear to be at odds. Yet, that gap can be bridged. Although the fixpoint semantics does not immediately resemble a general purpose loop, it does provide the semantics of a single `WHILE` loop. This is already sufficient to model *any* arbitrarily nested looping control flow [64]. However, the programs must be structured in a specific way, for example in *trampoline style* [10].

[64]: Solin (2011), ‘Normal forms in total correctness for while programs and action systems’

[10]: Ganz et al. (1999), ‘Trampoline style’

A program in trampoline style is organized as a single “scheduler” loop, the so-called *trampoline*, which manages all control flow. Execution of such programs proceeds in discrete steps. After each step, control is returned to the trampoline, which then proceeds to transfer control again [10]. This cycle continues until the program execution is finished. Using trampoline style, the program is effectively transformed into a state machine. We exploit the property, that only a single loop is required to express arbitrary control flow. This restricted form of control flow perfectly matches the semantics of SQL’s `WITH RECURSIVE` construct. The grammar of the trampoline-style ANF is shown in Figure 3.13.

p	::= fun $v(\overline{v} \ \tau)$: [SETOP] τ	top-level function
	{ f [f]}	
f	::= $v(\overline{v} \ \tau)$: $\tau = e$	function definition
e	::= a	expression
	<code>trampoline</code> (\overline{a})	trampoline invocation
	IF v THEN e ELSE e	
	LET $v = a$ IN e	
	EMIT a ; e	emit value
	CASE v OF ' v' ' : e	dispatching case
a	::= <code>SQL query</code> [\overline{v}]	boxed SQL query
v	::= \langle identifier \rangle	variable/function name
τ	::= \langle scalar SQL type \rangle	parameter/return type
t	::= <code>start</code> <code>trampoline</code>	trampoline function names

Figure 3.13.: Intermediate functional language in *trampoline* ANF.

If we apply trampoline style to the final ANF version of PL/SQL UDFs, we get the single-cycle call graph shown in Figure 3.14. All functions $f_i()$ are inlined into the `trampoline()` and pass control back to the dispatcher on function return, so `trampoline()` has full control over whether and how the computation continues. During execution, two parameters determine the transfer of control:

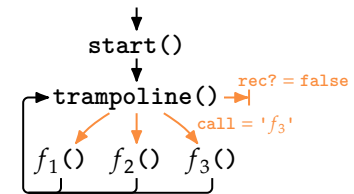


Figure 3.14.: Trampoline style.

`rec? ∈ {true, false}`: If parameter `rec?` is `false`, the trampoline will stop calculating and return.

`call ∈ { f_1, \dots, f_n }`: Otherwise, parameter `call` specifies the function to call next. When the function is finished, it returns control to the trampoline with new `rec?` and `call` values.

When we arrive at the pure SQL query after the final translation, these two control parameters and the state parameters are represented as a *row* in the working table of the recursive CTE. Because this row controls whether and how the `trampoline()` continues, we call these rows *control rows*.

To make this more concrete, Figure 3.15 shows a rewritten version of the ANF program in Figure 3.12 for UDF `route`. In this representation, the dispatcher `trampoline()` is central and provides the only way to exit the computation if the parameter `rec?` is `false` (Line 5). For the UDF `route`, there is only one function that the dispatcher can call (*i.e.*, there is only `call ∈ {while}`) because the control flow is very simple. Where the `while()` function previously returned a value to the caller, it now passes this return value to the trampoline as parameter `res` and parameter `rec? = false` to mark the computation as complete (see Lines 11 and 18). Recursive (self-)invocations are now achieved by passing the parameters `rec? = true`, `call = 'while'` to the `trampoline()` (see Line 22).

Note that *all* trampoline-style programs match the call graph shapes of Figures 3.14 and 3.15 and therefore fit into the restricted single-cycle iteration scheme implemented by `WITH RECURSIVE` (remember Figure 3.4). This is extremely powerful, because arbitrary programs of mutually recursive functions are susceptible to the trampoline transformation [10]. Therefore, trampoline style is a *universal* way to compile any iterative (and, as we will see in part “Functional Programming on Top of SQL Engines”, recursive) computation into

[10]: Ganz et al. (1999), ‘Trampoline style’

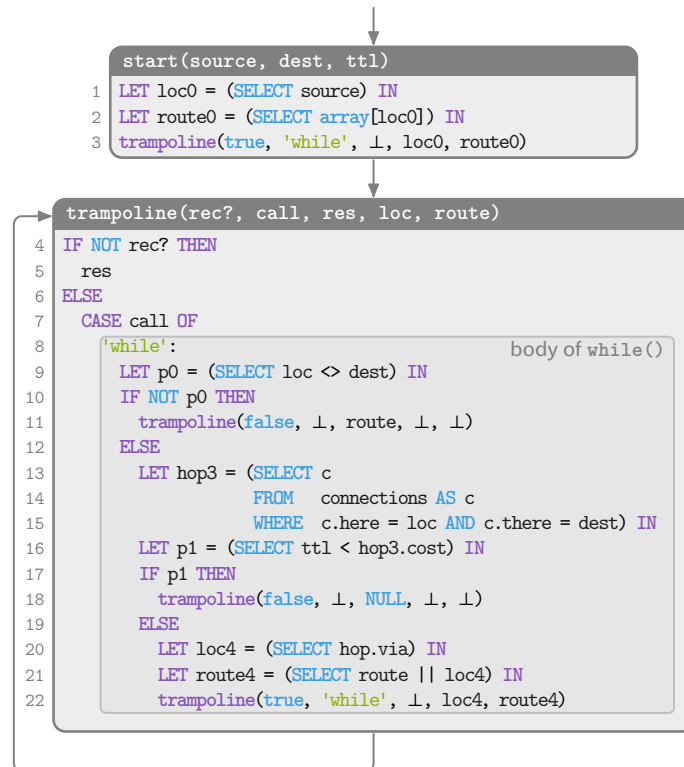


Figure 3.15. Program in trampolined style (ANF body of route inlined into trampoline). Compare to Figure 3.12.

SQL. This insight is fundamental to this line of research and a key contribution of this work.

3.6. Trampolined Style in SQL

The final stage of compilation is to convert the trampolined ANF into plain SQL. To do this, we must

1. translate the function body of `start()` into SQL `SELECT` block b_{start} , and `trampoline()` into SQL `SELECT` blocks $\text{tramp}_{0,\dots,n}$ and then
2. place these blocks in the `WITH RECURSIVE` based template that implements the trampoline. The instantiated template is the final output of the compiler.

The function bodies of `start()` and `trampoline()` consist of cascaded `LET` expressions, `IF-ELSE` conditionals, and `trampoline()` tail calls. The `trampoline()` function also contains the central `CASE OF` expression, which implements the dispatching of control to the appropriate function label. In step 1, a SQL `SELECT` block is constructed for each case of this `CASE OF` expression. The formal translation from trampolined style ANF to SQL is realized in terms of mapping $\mapsto_{\mathcal{A}}$ in Section 5.3.

Ramachandra et al. have pioneered a UDF compilation approach for branching yet linear, *non-looping* control flow, called `FROID` [27]. `FROID` compiles such UDFs into a single relational expression through a process called *algebraization*. This relational expression can then be inlined into the calling query, which essentially results in a (possibly

[27]: Ramachandra et al. (2018), ‘Froid: Optimization of Imperative Programs in a Relational Database’


```

1 WITH RECURSIVE run("rec?", call, res, ...) AS (
2   <b_start>
3   UNION ALL -- recursive UNION ALL
4   SELECT result.*
5   FROM run,
6   LATERAL (<tramp_0> WHERE run.call = 'tramp_0'           dispatcher
7            UNION ALL
8            :
9            UNION ALL
10           <tramp_n> WHERE run.call = 'tramp_n'
11          ) AS result
12 WHERE run."rec?"
13 )
14 SELECT run.res FROM run WHERE NOT run."rec?"

```

Figure 3.16.: SQL:1999-based trampoline template.

correlated) subquery. We largely adopt their strategy of compiling non-looping control flow. However, we generalize it. Instead of working with relational operators, we work exclusively with SQL. Also, instead of using the MICROSOFT SQL SERVER-specific `CROSS APPLY` [65], which they use to describe FROID's compilation approach, we use the SQL standard `LATERAL` join to express nested LET bindings. Additionally, we do not use `CASE WHEN` expressions for IF statements, we use predicated `UNION ALL` queries instead. We will discuss FROID in more detail in Section 7.2.

When evaluated, each block `b_start`, `tramp_0`, ..., `tramp_n` must return a **control row** to inform the dispatcher on how to proceed. The leading columns `"rec?"` and `call` determine the next block to jump to, or whether execution should terminate. The trailing columns contain the arguments that make up the state, as determined by the SSA to ANF conversion (see arguments `loc` and `route` in Figures 3.15 and 3.17). For scalar UDFs, the **control row** uses the `res` column to encode the result of the function call.

Step 2 embeds the `SELECT` blocks `b_start`, `tramp_0`, ..., `tramp_n` in the SQL-based trampoline template of Figure 3.16. The `b_start` `SELECT` block is placed in the non-recursive part of the recursive CTE. Remember that this will initialize the CTE (Section 3.1). The dispatcher is then evaluated repeatedly in Lines 6 to 10. In each iteration, the dispatcher uses the `call` column to select one `tramp_i` `SELECT` block for evaluation. Only the selected `SELECT` block places a new **control row** in the working table. Since all `SELECT` blocks must define new values for columns `call` and `"rec?"`, the dispatcher knows how to proceed in the next iteration.

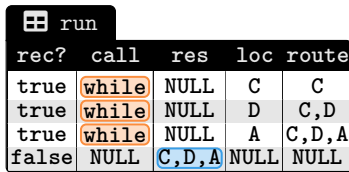
This iterative evaluation process continues until a `SELECT` block returns a **control row** in which the `"rec?"` column is `false`. The predicate in Line 12 will filter this row, resulting in an empty working table in the next iteration. This is the stop criterion for `WITH RECURSIVE`. The last **control row** and all of the rows collected before it are returned in the `UNION` table `run`. For scalar UDFs, the last **control row** contains the final result of the compiled UDF in the `res` column. This result is extracted in Line 14.

Instantiating the template completes the compilation process. Figure 3.17 shows the final trampolined style SQL query for the UDF `route` of Figure 2.2. Note that this query can be further optimized.

[65]: Galindo-Legaria et al. (2001), 'Orthogonal Optimization of Subqueries and Aggregation'

Variable `route` matches variable `res` perfectly, so one of them can be eliminated.

Whenever the original database application makes a call to the PL/SQL UDF `route`, the query code in Figure 3.17 Lines 2 to 27 can be used to replace that call. However, for convenience, the query is wrapped in a regular `LANGUAGE SQL` UDF. This allows us to easily replace the PL/SQL function without touching the application. Unfortunately, `POSTGRESQL`, for example, does not consider this query as simple enough to inline the code into the calling query. Therefore, replacing the function call with the plain SQL query is still required for best results.



rec?	call	res	loc	route
true	while	NULL	C	C
true	while	NULL	D	C,D
true	while	NULL	A	C,D,A
false	NULL	C,D,A	NULL	NULL

Table 3.1: UNION table computed by the recursive CTE in Figure 3.17 for call `route(C,A,10)`.

With the table instances in Figure 2.4, and the call `route(C,A,10)`, the recursive CTE computes the UNION table `run` in Table 3.1. This UNION table contains *the entire history* of the calculation steps. It shows that the `while()` function was called three times, and returned the partial paths (C), (C,D), and (C,D,A) before the final path `C,D,A` can be extracted. There was no context switching between the SQL executor and the PL/SQL interpreter to compute this result—in fact, no PL/SQL interpreter is required at all. The entire computation is performed by the SQL engine. This solves the problem of PL/SQL’s poor performance and scalability. SQL dialect differences aside, this compilation therefore allows PL/SQL UDFs to run on database engines that do not provide a PL/SQL interpreter.

```

1 CREATE FUNCTION route(source node, dest node, ttl int) RETURNS node[] AS $$
2   WITH RECURSIVE run("rec?", call, res, loc, route) AS (
3     SELECT true AS "rec?", 'while' AS call, NULL AS res, source AS loc, array[source] AS route
4     UNION ALL -- recursive UNION ALL
5     SELECT result.*
6     FROM   run,
7     LATERAL (SELECT if_p0.*
8              FROM   (SELECT loc <> dest) AS let_p0(p0),
9              LATERAL (SELECT false AS "rec?", NULL AS call, route AS res, NULL AS loc, NULL AS route
10             WHERE  NOT p0
11             UNION ALL
12             SELECT if_p1.*
13             FROM   (SELECT (SELECT c
14                        FROM   connections AS c
15                        WHERE  c.here = loc AND c.there = dest)) AS let_hop(hop),
16             LATERAL (SELECT ttl < hop.cost) AS let_p1(p1),
17             LATERAL
18             (SELECT false AS "rec?", NULL AS call, NULL AS res, NULL AS loc, NULL AS route
19             WHERE  p1
20             UNION ALL
21             SELECT true AS "rec?", NULL AS res, 'while' AS call, hop.via AS loc, route || hop.via AS route
22             WHERE  NOT p1) AS if_p1
23             WHERE  p0) AS if_p0
24             WHERE  run.call = 'while') AS result
25     WHERE  run."rec?"
26   )
27   SELECT run.res FROM run WHERE NOT run."rec?"
28 $$ LANGUAGE SQL;

```

Figure 3.17: The final plain SQL code output for the PL/SQL UDF `route` shown in Figure 2.2, complete with the instantiated trampoline.

Trampoline Style Manages Control, and Data Flow, Too

4.

In Chapter 3, we described a compilation method to transform *scalar* PL/SQL UDFs to a single recursive SQL CTE. While keeping the basic idea and compilation chain as is, we now separate the management of **control flow** and **data flow** to make the compilation suitable for *Table-valued functions (TVFs)*. To this end, we introduce the concept of **data rows** in addition to **control rows**. Previously, the compiler used only control rows which is not sufficient to handle TVFs.

In what follows, we describe how to adapt and generalize the CTE-based PL/SQL UDF compilation strategy to also cover TVFs like `march` of Figure 4.2a. We aim to (1) support the idiomatic `RETURN NEXT` style of UDF authoring, (2) avoid the materialization and copying of intermediate results, and (3) still avoid PL/SQL↔SQL switch overhead (which adds to 20% in the case of UDF `march`).

4.1. From Scalar Values To Tables

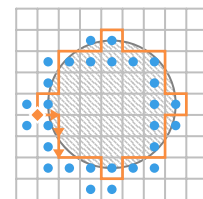
Let us look at an example. UDF `march` of Figure 4.2a is a TVF and implements the popular computer graphics algorithm *Marching Squares* [66] in PostgreSQL’s PL/PGSQL [36]. Whenever such a TVF encounters a `RETURN NEXT`, the PostgreSQL interpreter adds a new result to the function’s result set before the UDF resumes execution. In this example, `march(start)` returns a set of rows `•`, representing the contour for a two-dimensional object (see Figure 4.1a for an example).

To find this contour, `march` starts at an arbitrary 2×2 *contouring cell* \boxplus of pixels somewhere on the contour. Each contouring cell is stored as a row in table `squares`. The symbol \blacklozenge in Figure 4.1a marks the center of the start cell location. In each iteration of the `WHILE`-loop, Q_1 performs a join between the `directions` and `squares` tables to determine the direction to move in next. For instance, starting at \blacklozenge , the contouring cell \boxplus is encountered. The corresponding row in table `directions` specifies, that the current location (`cur`) moves one step to the right \rightarrow . In the next iteration, the cell configuration is \boxminus , therefore `cur` moves down \downarrow . The iteration continues until the start position is reached again. The movements in the `directions` table are carefully designed to prevent the algorithm from returning to a previously visited part of the contour.

During this *march* around the contour, the `RETURN NEXT` statement in Line 15 of Figure 4.2a just adds each visited location `cur` to the function’s materialized result set, rather than actually streaming it out of the function. This, potentially sizable, return set is materialized during execution and returned as a whole when the function exits [36].

[66]: Maple (2003), ‘Geometric design and space planning using the marching squares and marching cube algorithms’

[36]: PostgreSQL 15 PL/PGSQL Documentation



(a) Object.

directions				
\boxplus	\boxminus	\boxtimes	\boxminus	dir
\square	\square	\square	\square	\rightarrow
\square	\square	\blacksquare	\square	\uparrow
\square	\square	\blacksquare	\blacksquare	\rightarrow
\square	\blacksquare	\blacksquare	\blacksquare	\downarrow
\square	\blacksquare	\square	\blacksquare	\downarrow

(ten more rows)

(b) Directions table.

Figure 4.1: Sample object and directions-table.

```

1 CREATE FUNCTION march(start vec2) RETURNS SETOF vec2 AS $$
2 DECLARE
3   goal vec2 := start;
4   cur vec2 := start;
5   dir vec2;
6
7 BEGIN
8   WHILE true LOOP
9     dir := (SELECT d.dir
10            FROM directions AS d, squares AS s
11            WHERE s.xy = cur
12            AND (s.ll, s.lr, s.ul, s.ur)
13            = (d.ll, d.lr, d.ul, d.ur));
14
15     RETURN NEXT cur;
16
17     cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
18     EXIT WHEN cur = goal OR dir IS NULL;
19   END LOOP;
20   RETURN;
21 END;
22 $$ LANGUAGE PLPGSQL STRICT;

```

```

1 CREATE FUNCTION march-arr(start vec2) RETURNS vec2[] AS $$
2 DECLARE
3   goal vec2 := start;
4   cur vec2 := start;
5   dir vec2;
6   result vec2[] := ARRAY[] :: vec2[];
7 BEGIN
8   WHILE true LOOP
9     dir := (SELECT d.dir
10            FROM directions AS d, squares AS s
11            WHERE s.xy = cur
12            AND (s.ll, s.lr, s.ul, s.ur)
13            = (d.ll, d.lr, d.ul, d.ur));
14
15     result := result || cur;
16
17     cur := (cur.x + dir.x, cur.y + dir.y) :: vec2;
18     EXIT WHEN cur = goal OR dir IS NULL;
19   END LOOP;
20   RETURN result;
21 END;
22 $$ LANGUAGE PLPGSQL STRICT;

```

(a) Table-Valued version of PL/SQL UDF `march`.

(b) Marching Squares as an array-based PL/SQL UDF.

Figure 4.2.: $Q_1[\cdot]$ is an embedded SQL query with the free variable `cur`.

An alternative implementation as a *scalar* PL/SQL UDF (see `march-arr` in Figure 4.2b) iteratively builds the result as an array of type `vec2[]`. Our compilation strategy does handle `march-arr`, but the resulting SQL query will exhibit disappointing performance: the compilation creates a recursive CTE whose iteration expresses the iteration of the original UDF. This CTE maintains the local state of all UDF variables in a single row of the CTE’s working table. For UDF `march-arr`, maintaining the array `result` iteratively results in significant runtime overhead because the array has to be copied (and extended) in each iteration. For n iterations, this amounts to a total of $n \times (1 + 2 + \dots + (n - 1)) \cong \frac{1}{2}n^2 \times (n - 1)$ copy operations. In consequence, the CTE’s working table grows to 16 MB during the execution of the compiled UDF `march-arr`.

4.2. Control Flow Management

Recall that after compilation, each call to UDF `march` is encoded as a **control row** in the working table of `run`. This row determines the state of the machine, and thus which part of the computation to perform next. In Figure 4.3, each CFG construct that yields a control row is marked . The control row for each call is initially created in the non-recursive part of `run` (see Line 2 of Figure 4.5). In the recursive part of `run`, the row is read, because the two control columns `rec?` and `call` determine the transfer of control during execution.

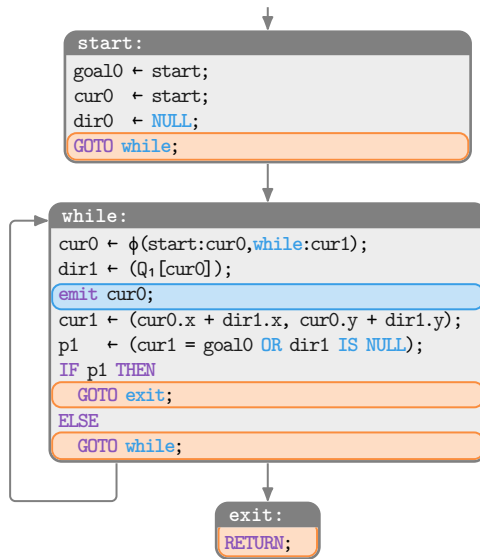


Figure 4.3.: CFG for UDF march with code blocks in SSA form.

The recursive part of `run` in Lines 4 to 23 of Figure 4.5 implements the dispatcher. Figure 4.4 depicts the central role of the dispatcher trampoline and how it realizes the control flow for UDF march.

The `call` column selects one function $\in \{\text{while}(), \text{exit}()\}$ for evaluation. All functions must return a new control row with columns "rec?" and `call`, so the dispatcher knows how to proceed in the next iteration (see Lines 13, 16 and 21 of Figure 4.5). This process continues until a block returns a control row with column "rec?"=false (see Line 21 of Figure 4.5). The working table in the next iteration will be empty, and `WITH RECURSIVE` evaluation stops. The trampolined-style SQL query thus implements the control flow of the original UDF march.

Reminder

rec?: If column `rec?` is false, the trampoline will stop calculating and return.
call: Otherwise, column `call` specifies the function to execute next. When the function is finished, it returns a control row to the trampoline with new `rec?` and `call` values.

4.3. Data Flow Management

While scalar UDFs return a single value in the last trampoline iteration, table-valued UDFs can return any number of values during execution (see `emit cur0` in Figure 4.3). The CTE of Figure 4.5 encodes these returned values in dedicated rows marked in Line 8 of Figure 4.5. Two columns manage this data flow:

data? $\in \{\text{true}, \text{false}\}$: Column `data?` indicates if this row has a valid return value in column `res`.
res: Contains this return value.

We call rows with column `data?=true` *data rows*. When a UDF uses either `RETURN NEXT` or `RETURN QUERY`, such data rows are created in addition to control rows.

Given the UDF of Figure 4.2a and assuming a call `march((8,7))`, overall the recursive CTE computes table `run` as shown on the next page. After the initialization, marked \blacklozenge , each iteration (separated by `--`) can generate two types of rows, data rows and control rows.

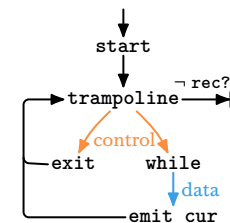


Figure 4.4.: Trampolined style.

```

1 WITH RECURSIVE run("rec?", "data?", call, res, cur) AS (
2   SELECT true AS "rec?", false AS "data?", 'while' AS call, NULL::vec2 AS res, start AS cur
3   UNION ALL -- recursive UNION ALL
4   SELECT result.*
5   FROM run,
6   LATERAL (SELECT if_p1.*
7             FROM ( Q,[run_cur] ) AS let_dir(dir),
8             LATERAL (SELECT NULL AS "rec?", true AS "data?", NULL AS call, run.cur AS res, NULL AS cur
9                       UNION ALL
10                      SELECT if_p2.*
11                      FROM (SELECT ((run.cur).x + dir.x, (run.cur).y + dir.y) :: vec2) AS let_cur(cur),
12                      LATERAL (SELECT let_cur.cur = start OR dir IS NULL) AS let_p1(p1),
13                      LATERAL (SELECT true AS "rec?", false AS "data?", 'while' AS call, NULL AS res, let_cur.cur AS cur
14                                WHERE NOT p1
15                                UNION ALL
16                                SELECT true AS "rec?", false AS "data?", 'exit' AS call, NULL AS res, NULL AS cur
17                                WHERE p1) AS if_p2
18             ) AS if_p1
19   WHERE run.call = 'while'
20   UNION ALL
21   SELECT false AS "rec?", false AS "data?", NULL AS call, NULL AS res, NULL AS cur
22   WHERE run.call = 'exit') AS result
23 WHERE run."rec?"
24 SELECT run.res FROM run WHERE run."rec?" IS NULL AND run."data?";

```

Figure 4.5.: Final plain SQL code emitted for the table-valued PL/SQL UDF march of Figure 4.2a.

(In general, any number of data rows can be created in each iteration.) Note how the last iteration indicates the end of execution via $(rec?, data?) = (false, false)$.

run				
rec?	data?	call	res	cur
true	false	while	NULL	(8,7)
false	true	NULL	(8,7)	NULL
true	false	while	NULL	(9,7)
false	true	NULL	(9,7)	NULL
false	true	NULL	(8,8)	NULL
true	false	exit	NULL	(8,8)
false	false	NULL	NULL	NULL

Recall that the original PL/SQL UDF has to materialize its table-valued result during execution, and returns all of it as a whole. This materialization prevents the surrounding execution plan from terminating prematurely, for example, when a `LIMIT` clause is used: `SELECT * FROM march((8,7)) LIMIT 5`. After compilation, these result values are immediately returned to the parent operator in terms of data rows, without having to materialize the entire result. This saves memory and reduces the runtime. In addition, important metrics such as CPU cost and cardinalities can be estimated more accurately, making planning of the translation much more effective: While PL/SQL UDFs are effectively a black box for the planner, the translation is a regular SQL query that the planner is designed to handle.

4.4. The Impact of Data Rows in Trampoline Style SQL

Both UDFs, `march` and `march-arr`, indeed exhibit the infamous context switching overhead that gives PL/SQL programming its bad reputation. We have measured that the back and forth between PL/SQL and SQL accounts for 20% of the overall evaluation time for both variants (see Table 4.1). The compilation to recursive SQL CTEs described in [13, 15] avoids this particular overhead for the two UDFs.

[13]: Duta et al. (2020), ‘Compiling PL/SQL Away’

[15]: Hirn et al. (2021), ‘One WITH RECURSIVE is Worth Many GOTOs’

However, the naive treatment of the iterative `result` array construction and copying in the CTE for the scalar UDF `march-arr` quickly eats up all the gains: the quadratic array maintenance costs mentioned at the beginning of this chapter add up to about 50% of the

UDF	Return Type	Overhead	Runtime	Memory
march-arr	vec2[]	20%	112.8% (0.88×)	16 MB
march	SETOF vec2	20%	38.2% (2.61×)	110 kB

Table 4.1: The context switching overhead before and speedup as well as working table size after compilation.

overall CTE runtime. If we double the size of UDF input, the working table of the CTE for `march-arr` grows by a factor of four (from 16 MB to 64 MB) and the array maintenance overhead increases to 56%. Ultimately, this leads to a *slowdown* of `march-arr` after compilation.

In stark contrast, the control- and data-flow-aware compilation strategy sketched in Section 3.5, translates the table-valued UDF into the recursive SQL CTE of Figure 4.5. Array construction and copying is avoided altogether and the working table size remains small: doubling the UDF input size—and thus the number of iterations performed—linearly grows the working table’s size from 110 kB to a mere 220 kB. Overall, compilation of UDF `march` leads to runtime reduction of 62% (*i. e.*, post-compilation the UDF runs about 2.6 times faster). In addition, materialization is entirely avoided: the CTE of Figure 4.5 can stream the rows of the resulting table to the downstream plan. This is a major advantage over the PL/SQL UDF that materializes the entire result before returning it.

From PL/SQL to SQL: Behind the Scenes

5.

Four syntactic transformations are part of the PL/SQL UDF compiler implementation. This chapter describes these transformations in detail, using syntax-to-syntax mappings \mapsto_s , \mapsto_A , \mapsto_f , and \mapsto_Q . We have chosen to use *inference rules* to define these mappings according to syntactic cases: the case (or consequence) below the line follows if the antecedents above the line are satisfied. Read these rules in order ①, ②, ③, as shown in Figure 5.1.

Throughout the inference rules, we will use color ■ to highlight syntax elements of the input language, color ■ for the output language, and color ■ is used for SQL fragments.

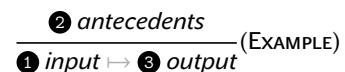


Figure 5.1.: Inference rule reading order.

5.1. From PL/SQL to SSA

The set of inference rules for compiling from PL/SQL to SSA consists of \mapsto_s , and auxiliaries \ddagger and \leftrightarrow . Compilation starts with the top-level rule UDF, which takes a PL/SQL UDF as input and returns a GOTO-based program. The subsequent rules map the body of the PL/SQL UDF into a dictionary s of blocks. These blocks contain statements of the simple imperative GOTO-based form defined in Section 3.3. Within the dictionary s , each block is identified by its label κ . The core of these rules is the relation $\Gamma \vdash \langle c \mid \kappa_1 \mid s_1 \rangle \mapsto_s \langle \kappa_2 \mid s_2 \rangle$, which defines the transformation of single PL/SQL statements c into a sequence of simple imperative statements. These simple statements are appended to the statements in the block labeled κ_1 . The old *label-to-block* dictionary s_1 is updated to s_2 via $s_2 \equiv s_1 +_{\kappa} [\langle \text{statements} \rangle]$, which creates block κ_1 in s_2 if it does not already exist in s_1 . Once c has been translated, subsequent statements are to be appended to block κ_2 . The auxiliary \ddagger uses the continuation block label κ_2 to compile entire sequences of PL/SQL statements (see Rules SEQ and SEQ0).

Auxiliary \leftrightarrow (Rule ITER) is used to compile any form of iteration like LOOP, WHILE, and FOR statements. For such statements, rule \mapsto_s uses auxiliary \leftrightarrow to translate the loop body, and to create the necessary block labels which are returned as a quadruple $(\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end})$. Rules LOOP, WHILE, and FOR use these block labels to establish looping control flow as shown in Figure 5.2. This block arrangement simplifies the compilation of PL/SQL's CONTINUE and EXIT statements in terms of the imperative statements GOTO κ_{head} and GOTO κ_{end} , respectively (see Rules EXR and CONT). Because PL/SQL loops can be nested, all rules pass or maintain a stack Γ of $\langle \kappa_{head}, \kappa_{end} \rangle$, the top entry of which refers to the current innermost loop.

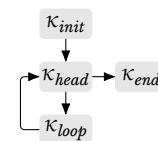


Figure 5.2.: Block labels produced by auxiliary \leftrightarrow for arbitrary looping control flow.

The inference rules produce a program that is not yet in SSA form. However, as already mentioned in Section 3.3, this is a straightforward process following standard algorithms [52, 54, 57]. Since we are aiming for crude SSA, this is even easier to achieve.

[52]: Cytron et al. (1991), 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph'

[54]: Braun et al. (2013), 'Simple and Efficient Construction of Static Single Assignment Form'

[57]: Appel (1998), 'SSA is Functional Programming'

Function Definition.

$$\frac{\emptyset \vdash \langle p \mid \text{start} \mid [] \rangle \mapsto_{\mathcal{S}} \langle \kappa_1 \mid s_1 \rangle \quad \text{blocks} \equiv \bigcup_{\kappa \in \mathcal{S}_1} \kappa : \varepsilon \ s_1[\kappa]}{\vdash \text{CREATE FUNCTION } v(v_0 \ \tau_0, \dots, v_n \ \tau_n) \text{ RETURNS [SETOF]} \ \tau_r \text{ AS } \text{\$ \$ \$ } p \ \text{\$ \$ \$ } \text{ LANGUAGE PLPGSQL STABLE; } \mapsto_{\mathcal{S}} \text{fun } v(v_0 \ \tau_0, \dots, v_n \ \tau_n) : \text{[SETOF]} \ \tau_r \{ \text{blocks} \}} \text{(UDF)}$$

$$\frac{\Gamma \vdash \langle \text{vars} \mid \kappa \mid s \rangle \ ; \ \langle \kappa_1 \mid s_1 \rangle \quad \Gamma \vdash \langle \text{stmts} \mid \kappa_1 \mid s_1 \rangle \ ; \ \langle \kappa_2 \mid s_2 \rangle}{\Gamma \vdash \langle \text{DECLARE vars BEGIN stmts END} \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa_2 \mid s_2 \rangle} \text{(BODY)}$$

Reminder

```

f ::= CREATE FUNCTION v( $\overline{v \ \tau}$ )
      RETURNS [SETOF]  $\tau$  AS  $\text{\$ \$ \$ } p \ \text{\$ \$ \$ }$ 
      LANGUAGE PLPGSQL STABLE;
p ::= DECLARE d; BEGIN s; END
d ::=  $v \ \tau \mid v \ \tau := a \mid d; d$ 
s ::=  $v := a$ 
      | IF a THEN s; [ELSE s;] END IF
      | LOOP s; END LOOP
      | WHILE a LOOP s; END LOOP
      | FOR v IN a..a LOOP s; END LOOP
      | EXIT | CONTINUE | RETURN a
      | RETURN NEXT a | RETURN QUERY a
      | s; s
a ::= SQL query [ $\overline{v}$ ]
v ::= (identifier)
 $\tau$  ::= (scalar SQL type)

```

```

f ::= fun v( $\overline{v \ \tau}$ ) : [SETOF]  $\tau$  { $\overline{b}$ }
b ::=  $\kappa : p \ s;$ 
p ::=  $v : \tau \ \vdash \ \phi(\kappa : \overline{v}); p$ 
      |  $\varepsilon$ 
s ::=  $v \ \vdash \ a$ 
      | IF v THEN t ELSE t
      | t
      | EMIT a
      | s; s
t ::= GOTO  $\kappa$ 
      | RETURN a
a ::= SQL query [ $\overline{v}$ ]
v ::= (identifier)
 $\tau$  ::= (scalar SQL type)
 $\kappa$  ::= (block label)

```

Statement Sequences.

$$\frac{\Gamma \vdash \langle \text{stmt} \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa_1 \mid s_1 \rangle \quad \Gamma \vdash \langle \text{stmts} \mid \kappa_1 \mid s_1 \rangle \ ; \ \langle \kappa_2 \mid s_2 \rangle}{\Gamma \vdash \langle \text{stmt; stmts} \mid \kappa \mid s \rangle \ ; \ \langle \kappa_2 \mid s_2 \rangle} \text{(SEQ)}$$

$$\frac{}{\Gamma \vdash \langle \varepsilon \mid \kappa \mid s \rangle \ ; \ \langle \kappa \mid s \rangle} \text{(SEQO)}$$

Statements.

$$\frac{s_1 \equiv s +_{\kappa} [v \ \vdash \ q;]}{\Gamma \vdash \langle v \ \tau := q \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa \mid s_1 \rangle} \text{(DECL)}$$

$$\frac{s_1 \equiv s +_{\kappa} [v \ \vdash \ (\text{SELECT NULL});]}{\Gamma \vdash \langle v \ \tau \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa \mid s_1 \rangle} \text{(DECLO)}$$

$$\frac{s_1 \equiv s +_{\kappa} [v \ \vdash \ q;]}{\Gamma \vdash \langle v := q \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa \mid s_1 \rangle} \text{(ASSIGN)}$$

Loops.

$$\frac{\begin{array}{l} \kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end} \equiv \text{new block labels} \\ \Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash \langle \text{stmts} \mid \kappa_{body} \mid s \rangle \ ; \ \langle \kappa_1 \mid s_1 \rangle \quad s_2 \equiv s_1 +_{\kappa} [\text{GOTO } \kappa_{init};] \end{array}}{\Gamma \vdash \langle \text{stmts} \mid \kappa \mid s \rangle \rightsquigarrow \left((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}) \mid \kappa_1 \mid s_2 \right)} \text{(ITER)}$$

$$\frac{\begin{array}{l} \Gamma \vdash \langle \text{stmts} \mid \kappa \mid s \rangle \rightsquigarrow \left((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}) \mid \kappa_1 \mid s_1 \right) \\ s_2 \equiv s_1 +_{\kappa_{init}} [\text{GOTO } \kappa_{init};] +_{\kappa_{head}} [\text{GOTO } \kappa_{body};] +_{\kappa_1} [\text{GOTO } \kappa_{head};] \end{array}}{\Gamma \vdash \langle \text{LOOP stmts; END LOOP} \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa_{end} \mid s_2 \rangle} \text{(LOOP)}$$

$$\frac{\begin{array}{l} \Gamma \vdash \langle \text{stmts} \mid \kappa \mid s \rangle \rightsquigarrow \left((\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end}) \mid \kappa_1 \mid s_1 \right) \quad p \equiv \text{new var} \\ b \equiv [p \ \vdash \ q; , \text{IF } p \ \text{THEN GOTO } \kappa_{body} \ \text{ELSE GOTO } \kappa_{end};] \\ s_2 \equiv s_1 +_{\kappa_{init}} [\text{GOTO } \kappa_{head};] +_{\kappa_{head}} b +_{\kappa_1} [\text{GOTO } \kappa_{head};] \end{array}}{\Gamma \vdash \langle \text{WHILE } q \ \text{LOOP stmts; END LOOP} \mid \kappa \mid s \rangle \mapsto_{\mathcal{S}} \langle \kappa_{end} \mid s_2 \rangle} \text{(WHILE)}$$

$$\frac{\Gamma \vdash \langle stmts \mid \kappa \mid s \rangle \rightsquigarrow \left\langle \left(\kappa_{init}, \kappa_{head}, \kappa_{body}, \kappa_{end} \right) \mid \kappa_1 \mid s_1 \right\rangle \quad p, v_1 \equiv \text{new vars} \quad \begin{array}{l} b_0 \equiv [v \leftarrow q_0; \text{GOTO } \kappa_{head};] \\ b_1 \equiv [v_1 \leftarrow q_1; p \leftarrow v <= v_1; \text{IF } p \text{ THEN GOTO } \kappa_{body} \text{ ELSE GOTO } \kappa_{end};] \\ s_2 \equiv s_1 +_{\kappa_{init}} b_0 +_{\kappa_{head}} b_1 +_{\kappa_1} [v \leftarrow v + 1; \text{GOTO } \kappa_{head};] \end{array}}{\Gamma \vdash \langle \text{FOR } v \text{ IN } q_0 \cdot q_1 \text{ LOOP } stmts; \text{END LOOP} \mid \kappa \mid s \rangle \mapsto_s \langle \kappa_{end} \mid s_2 \rangle} \text{(FOR)}$$

$$\frac{s_1 \equiv s +_{\kappa} [\text{GOTO } \kappa_{end};]}{\Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash \langle \text{EXIT} \mid \kappa \mid s \rangle \mapsto_s \langle \kappa \mid s_1 \rangle} \text{(EXIT)}$$

$$\frac{s_1 \equiv s +_{\kappa} [\text{GOTO } \kappa_{head};]}{\Gamma, \langle \kappa_{head}, \kappa_{end} \rangle \vdash \langle \text{CONTINUE} \mid \kappa \mid s \rangle \mapsto_s \langle \kappa \mid s_1 \rangle} \text{(CONT)}$$

Linear Control Flow.

$$\frac{\kappa_{then}, \kappa_{else}, \kappa_{meet} \equiv \text{new block labels} \quad p \equiv \text{new var} \quad \Gamma \vdash \langle stmts_1 \mid \kappa_{then} \mid s \rangle \circ \langle \kappa_1 \mid s_1 \rangle \quad \Gamma \vdash \langle stmts_2 \mid \kappa_{else} \mid s_1 \rangle \circ \langle \kappa_2 \mid s_2 \rangle \quad \begin{array}{l} b \equiv [p \leftarrow q; \text{IF } p \text{ THEN GOTO } \kappa_{then} \text{ ELSE GOTO } \kappa_{else};] \\ s_3 \equiv s_2 +_{\kappa} b +_{\kappa_1} [\text{GOTO } \kappa_{meet};] +_{\kappa_2} [\text{GOTO } \kappa_{meet};] \end{array}}{\Gamma \vdash \langle \text{IF } q \text{ THEN } stmts_1; \text{ELSE } stmts_2; \text{END IF} \mid \kappa \mid s \rangle \mapsto_s \langle \kappa_{meet} \mid s_3 \rangle} \text{(IFELSE)}$$

$$\frac{\Gamma \vdash \langle \text{IF } q \text{ THEN } stmts; \text{ELSE } \varepsilon \text{ END IF} \mid \kappa \mid s \rangle \mapsto_s \langle \kappa_1 \mid s_1 \rangle}{\Gamma \vdash \langle \text{IF } q \text{ THEN } stmts; \text{END IF} \mid \kappa \mid s \rangle \mapsto_s \langle \kappa_1 \mid s_1 \rangle} \text{(IF)}$$

$$\frac{s_1 \equiv s +_{\kappa} [\text{RETURN } q;]}{\Gamma \vdash \langle \text{RETURN } q \mid \kappa \mid s \rangle \mapsto_s \langle \kappa \mid s_1 \rangle} \text{(RETURN)}$$

5.2. From SSA to ANF

This set of inference rules is an adaptation of the work of Chakravarty, Keller, and Zadarnowski published in [9]. Since we know that the input is always in crude SSA, we can specialize the algorithm to our grammar and thus simplify it. Rules \mapsto_A and auxiliary \Rightarrow_A implement all necessary translations. The translation is mostly straightforward. However, **GOTO** statements require that we first create a data structure that we can use to store argument lists for translated **GOTO**s. Auxiliary \Rightarrow_A traverses all ϕ -functions of all blocks, and collects a set of triples ($\langle \text{from } \kappa \rangle, \langle \text{to } \kappa \rangle, \langle \text{arguments} \rangle$). During translation, a lookup in this data structure suffices to determine the function call arguments. For the example in Figure 5.3, \Rightarrow_A constructs the following set: $\{\kappa_1, \kappa_1, [c_1, \dots, c_n]\}$. This information is then used in Rule **GOTO** to compile **GOTO** κ_1 to $\kappa_1(c_1, \dots, c_n)$.

The second auxiliary \Rightarrow_A (Rules **PHI** and **PHI EMPTY**) traverses the ϕ -functions of one block and collects the names v and types τ . This information is then used to create the parameter list of function κ (ps). See Figure 5.3 for an example. Auxiliary \Rightarrow_A extracts $v_1 : \tau_1, \dots, v_n : \tau_n$ to create function signature $\kappa_1(v_1 : \tau_1, \dots, v_n : \tau_n)$.

[9]: Chakravarty et al. (2004), ‘A functional perspective on SSA optimisation algorithms’

```

1 fun v(v τ, ..., v τ) : τ {
2   κ1:
3     v1:τ1 ← φ(..., κ1:c1, ...);
4     ⋮
5     vn:τn ← φ(..., κn:cn, ...);
6     ⋮
7     GOTO κ1;
8 }
```

```

1 fun v(v τ, ..., v τ) : τ {
2   κ1(v1:τ1, ..., vn:τn):
3     ⋮
4     κ1(c1, ..., cn);
5 }
```

Figure 5.3.: The top code is in SSA form, the bottom code is in ANF form.

The Translation Function.

$$\frac{\vdash \langle \kappa_j : \mathbf{b}_i \mid \emptyset \rangle \mapsto_{\mathcal{A}} V_i \mid_{i=0 \dots n} \quad V \equiv \bigcup_{i=0 \dots n} V_i}{V \vdash \mathbf{b}_i \mapsto_b \mathbf{f}_i \mid_{i=0 \dots n}} \text{(FUNCTION)}$$

$$\text{fun } v(\overline{v \tau}) : \tau \quad \{ \kappa_0 : \mathbf{b}_0; \dots; \kappa_n : \mathbf{b}_n \} \mapsto_{\mathcal{A}} \text{fun } v(\overline{v \tau}) : \tau \quad \{ \mathbf{f}_0 \dots \mathbf{f}_n \}$$

Reminder

```

f ::= fun v(̄v τ) : [SETOF] τ {̄b}
b ::= κ : p s;
p ::= v: τ ← ϕ(κ:̄v); p
   | ε
s ::= v ← a
   | IF v THEN t ELSE t
   | t
   | EMIT a
   | s; s
t ::= GOTO κ
   | RETURN a
a ::= SQL query [̄v]
v ::= identifier
τ ::= scalar SQL type
κ ::= block label

```

```

p ::= fun v(̄v τ) : [SETOF] τ {̄f}
f ::= v(̄v τ) : τ = e
e ::= a
   | v(̄a)
   | IF v THEN e ELSE e
   | LET v = a IN e
   | EMIT a; e
a ::= SQL query [̄v]
v ::= identifier
τ ::= scalar SQL type

```

Convert a Block Into a Function.

$$\frac{\langle \Gamma, \kappa \rangle \vdash p \Rightarrow_{\mathcal{A}} \mathbf{ps} \quad \langle \Gamma, \kappa \rangle \vdash s \mapsto_{\mathcal{A}} s_1}{\Gamma \vdash \kappa : p \ s; \mapsto_b \kappa(\mathbf{ps}) = s_1} \text{(BLOCK)}$$

Convert SSA Statements Into ANF Expressions.

$$\frac{\Gamma \vdash s \mapsto_{\mathcal{A}} s_1}{\Gamma \vdash v \leftarrow a; s \mapsto_{\mathcal{A}} \text{LET } v = a \text{ IN } s_1} \text{(ASSIGN)}$$

$$\frac{\Gamma \vdash t_0 \mapsto_{\mathcal{A}} e_0 \quad \Gamma \vdash t_1 \mapsto_{\mathcal{A}} e_1}{\Gamma \vdash \text{IF } v \text{ THEN } t_0 \text{ ELSE } t_1 \mapsto_{\mathcal{A}} \text{IF } v \text{ THEN } e_0 \text{ ELSE } e_1} \text{(IF)}$$

$$\frac{\Gamma \vdash s \mapsto_{\mathcal{A}} s_0}{\Gamma \vdash \text{EMIT } a; s \mapsto_{\mathcal{A}} \text{EMIT } a; s_0} \text{(EMIT)}$$

Convert Block Terminals Into ANF Expressions.

$$\frac{p \equiv \{v \mid (\kappa_0, \kappa_1, v) \in \Gamma \wedge \kappa_0 = \kappa_t \wedge \kappa_1 = \kappa\}}{\langle \Gamma, \kappa_t \rangle \vdash \text{GOTO } \kappa \mapsto_{\mathcal{A}} \kappa(p)} \text{(GOTO)}$$

$$\frac{}{\Gamma \vdash \text{RETURN } a \mapsto_{\mathcal{A}} a} \text{(RETURN)}$$

Find Function Parameters Based on ϕ Functions.

$$\frac{\Gamma \vdash p \Rightarrow_{\mathcal{A}} e}{\Gamma \vdash v: \tau \leftarrow \phi(\dots); p \Rightarrow_{\mathcal{A}} v: \tau \ \diamond \ e} \text{(PHI)}$$

$$\frac{}{\Gamma \vdash \varepsilon \Rightarrow_{\mathcal{A}} \phi} \text{(PHI EMPTY)}$$

Prepare Argument Lists for Translated GOTOS.

$$\frac{\text{pred} \equiv \{(\kappa, \kappa_0, v) \mid \kappa_0 : v \in P\} \quad \Gamma \vdash \langle \kappa \mid p \mid V \ \diamond \ \text{pred} \rangle \Rightarrow_{\mathcal{A}} p_0}{\Gamma \vdash \langle \kappa \mid v \leftarrow \phi(P) \rangle; p \mid V \rangle \Rightarrow_{\mathcal{A}} p_0} \text{(CALLEE)}$$

$$\frac{\mathbf{g} \equiv \left\{ \left(\kappa, \kappa_0, \left\{ v \mid (\kappa_1, \kappa_2, v) \in V \wedge \kappa = \kappa_1 \wedge \kappa_0 = \kappa_2 \right\} \right) \mid (\kappa, \kappa_0, v_0) \in V \right\}}{\Gamma \vdash \langle \kappa \mid \varepsilon \mid V \rangle \Rightarrow_{\mathcal{A}} \mathbf{g}} \text{(CALLEE EMPTY)}$$

5.3. From ANF to Trampoline Style ANF

This set of inference rules \mapsto_J transforms the ANF input program into trampolined-style ANF. The top-level function signature remains the same. The mutually recursive function family $f_0 \dots f_n$ is eliminated in this process (see Rule FUNCTION). Only the start and trampoline functions remain. The start function is the entry point of the program. It calls the trampoline with the initial arguments. The trampoline function is responsible for dispatching the execution to the correct function based on the function label argument.

Rules TRAMPOLINE and RETURN transform every call to a function $f_0 \dots f_n$, and every return of a value into a call to the now central trampoline. There are no structural changes to the other syntax elements (see Rules IFELSE, LET, and EMIT).

The Translation Rules.

$$\begin{array}{c}
 f_i \mapsto_J b_i \mid_{i=0..n} \quad f_{\text{start}} \equiv \text{start}(\overline{v \ \tau}) : \tau = b_0 \\
 \text{params} \equiv \text{"rec?" BOOL, call TEXT, res } \tau, \text{"data?" BOOL, } \overline{v \ \tau} \\
 \\
 \begin{array}{c}
 \text{trampoline(params) : } \tau = \\
 \text{IF NOT "rec?" THEN} \\
 \text{res} \\
 \text{ELSE} \\
 \text{CASE call OF} \\
 \text{'f}_1\text{' : } b_1 \\
 \vdots \\
 \text{'f}_n\text{' : } b_n
 \end{array} \\
 \\
 \text{f}_{\text{tramp}} \equiv \text{trampoline(params) : } \tau = \dots \\
 \\
 \frac{}{\text{fun } \overline{v \ \tau} : [\text{SETOF}] \ \tau \quad \{ f_0 \dots f_n \} \mapsto_J \text{fun } \overline{v \ \tau} : [\text{SETOF}] \ \tau \quad \{ f_{\text{start}} \ f_{\text{tramp}} \}} \text{(FUNCTION)}
 \end{array}$$

Reminder

```

p ::= fun v(τ) : [SETOF] τ {f}
f ::= v(τ) : τ = e
e ::= a
    | v(ā)
    | IF v THEN e ELSE e
    | LET v = a IN e
    | EMIT a; e
a ::= SQL query [v]
v ::= <identifier>
τ ::= <scalar SQL type>

```

```

p ::= fun v(τ) : [SETOF] τ {f [f]}
f ::= v(τ) : τ = e
e ::= a
    | trampoline(ā)
    | IF v THEN e ELSE e
    | LET v = a IN e
    | EMIT a; e
    | CASE v OF 'v' : e
a ::= SQL query [v]
v ::= <identifier>
τ ::= <scalar SQL type>
t ::= start | trampoline

```

Trampoline Calls and Returns.

$$\frac{}{v(\overline{a}) \mapsto_J \text{trampoline}(\text{true}, \overline{v'}, \text{NULL}, \text{false}, \overline{a})} \text{(TRAMPOLINE)}$$

$$\frac{}{a \mapsto_J \text{trampoline}(\text{false}, \text{NULL}, a, \text{true}, \text{NULL})} \text{(RETURN)}$$

Other ANF Expressions.

$$\frac{e_1 \mapsto_J t_1 \quad e_2 \mapsto_J t_2}{\text{IF } v \text{ THEN } e_1 \text{ ELSE } e_2 \mapsto_J \text{IF } v \text{ THEN } t_1 \text{ ELSE } t_2} \text{(IFELSE)}$$

$$\frac{e \mapsto_J t}{\text{LET } v = a \text{ IN } e \mapsto_J \text{LET } v = a \text{ IN } t} \text{(LET)}$$

$$\frac{e \mapsto_J t}{\text{EMIT } a; e \mapsto_J \text{EMIT } a; t} \text{(EMIT)}$$

5.4. From Trampolined Style ANF to SQL

The last set of inference rules \mapsto_Q transforms programs in trampolined style ANF to the final SQL query. The top-level Rule `NONREC` is used for linear, non-looping programs. In this case, there is no need for a trampoline and therefore no need for a recursive CTE. Otherwise, Rule `REC` instantiates the `WITH RECURSIVE`-based template for trampolined-style programs.

We use $e_i \mapsto_Q (q_i, t_i)$ to translate expression e_i into a list of SQL tables q_i . Each of these tables can be referenced by its unique row variable t_i . To read the value of e_i , we can simply use `SELECT t_i.* FROM q_i`.

- ▶ Rule `EMBED` handles blackboxed subexpressions `Q[v1, ..., vn]`. Such subexpressions may include variable references, entire SQL queries, or the use of built-in functions and operators. The rule substitutes the free variables in `Q` with the appropriate parameters v_1, \dots, v_n . The resulting SQL expression is wrapped in a simple `SELECT` to ensure proper behavior with empty results. In this case, `(SELECT q)` ensures that the empty result is converted to a `NULL` value. This is important because this expression is used as part of a join in the `FROM` clause of the final translation. Since we are using simple `LATERAL` joins to chain these expressions, a single empty result means that the entire join would be empty. This would cause the execution to terminate prematurely.
- ▶ Rule `CALL` handles all trampoline interactions, and generally produces a `control row` that encodes how to proceed. The `control row` contains the function label l , and the arguments a_1, \dots, a_n . This row is then returned to the trampoline which then uses this information to dispatch the execution to the correct function. The `control row` is also used to determine whether the execution should terminate.
- ▶ Rule `LET` translates all variable bindings `LET v = a IN e`. To map this semantics to SQL, the expression a must first be evaluated and its value bound to v , then the expression e is evaluated. A `LATERAL` join is perfectly in line with this evaluation semantics [38, 65]. Nested `LET`s and SQL's row variable visibility coincide. That is, whatever is bound to the left of `LATERAL` is visible to the right.
- ▶ Rule `COND` compiles `IF v THEN e ELSE e` conditionals in terms of SQL's `UNION ALL` clause. The `WHERE` predicates p_i are independent of their respective query blocks b_i . This is important because it allows us to use SQL's `RESULT` operator to implement lazy evaluation of conditionals (see Paragraph Multi-way Conditionals).
- ▶ Rule `EMIT` is used for table-valued UDFs. This rule constructs a `data row` for the emitted value a , which is returned to the trampoline in addition to the `control row`. Again, we use SQL's `UNION ALL` clause to combine these rows into one result. This is possible because the `control row` and the `data row` have the same schema.

[38]: *SQL:1999 Standard. Database Languages—SQL—Part 2: Foundation*

[65]: Galindo-Legaria et al. (2001), 'Orthogonal Optimization of Subqueries and Aggregation'

Multi-way Conditionals. Rules COND and CASE compile conditional statements into SQL. These rules use a stack of $n - 1$ UNION ALLs to implement the case distinction between query blocks b_1, \dots, b_n . These query blocks contain mutually exclusive WHERE predicates p_i that are *independent* of their block. See Rule CASE, for example. Here, predicates WHERE call='v_i' do not depend on values produced by b_i . For PostgreSQL, this translates to the physical plan in Figure 5.4. The RESULT operator evaluates the *one-time filter* p_i once *before* processing the subplan for block b_i [33]. If the predicate p_i is evaluated to be *false*, then the plan for b_i will never be executed. This execution behavior exactly implements the expected laziness of IF ELSE or CASE OF conditionals. We have found this to be a compositional and performant translation of conditionals on PostgreSQL. Other RDBMSs implement similar operator configurations and runtime behavior. For example, in ORACLE [67], a UNION ALL/FILTER pair ensures that *either* b_1 or b_2 is evaluated.

[33]: PostgreSQL 15 Documentation

[67]: Oracle Database PL/SQL Language Reference 21c

However, there are exceptions. DuckDB [68], for example, implements a push-based execution model, not a pull-based one. In short, this means that execution is bottom-up, not top-down as is the case in systems using the *volcano iterator model*. Therefore, in such systems, the plans for b_1 and b_2 are evaluated before a FILTER or RESULT node can prevent execution. As a result, the execution is not lazy. Local changes to the emitted SQL code by \mapsto_Q should be able to accommodate the specifics of a wide range of RDBMS targets. For example, instead of using the predicated UNION ALL branches, it may be sufficient to use CASE WHEN expressions to get lazy execution of conditionals.

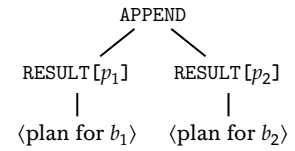


Figure 5.4.: Physical plan on PostgreSQL for conditionals.

[68]: Raasveldt et al. (2019), ‘DuckDB: an Embeddable Analytical Database’

To SQL Rules.

	$e_{\text{start}} \mapsto_Q \langle q_{\text{start}} \mid t \rangle$	(NONREC)
<pre>fun v($\bar{v} \bar{\tau}$) : [SETOF] τ { f_start($\bar{v} \bar{\tau}$) : $\tau_1 = e_{\text{start}}$ }</pre>	\mapsto_Q	<pre>CREATE FUNCTION v($\bar{v} \bar{\tau}$) RETURNS [SETOF] τ AS \$\$ SELECT t.* FROM q_start \$\$ LANGUAGE SQL;</pre>
	$e_{\text{start}} \mapsto_Q \langle q_{\text{start}} \mid t_{\text{start}} \rangle$ $e_{\text{tramp}} \mapsto_Q \langle q_{\text{tramp}} \mid t_{\text{tramp}} \rangle$	(REC)
<pre>fun v($\bar{v} \bar{\tau}$) : [SETOF] τ { f_start($\bar{v} \bar{\tau}$) : $\tau_1 = e_{\text{start}}$ f_tramp($\bar{v} \bar{\tau}$) : $\tau_2 = e_{\text{tramp}}$ }</pre>	\mapsto_Q	<pre>CREATE FUNCTION v($\bar{v} \bar{\tau}$) RETURNS [SETOF] τ AS \$\$ WITH RECURSIVE run("rec?", call, res, "data?", $\bar{v} \bar{\tau}$) AS (SELECT f_start.* FROM q_start UNION ALL -- recursive UNION SELECT f_tramp.* FROM run, LATERAL q_tramp WHERE run."rec?") SELECT run.res FROM run WHERE NOT run."rec?" AND run."data?" \$\$ LANGUAGE SQL;</pre>

Trampoline Call to SQL Row.

$$\frac{t \equiv \text{fresh row var} \quad q \equiv (\text{SELECT } a_1, \dots, a_n \text{ AS } t)}{\text{trampoline}(\bar{a}) \mapsto_Q \langle q \mid t \rangle} \text{(CALL)}$$

Other Trampolined ANF Expressions.

$$\frac{a \mapsto_Q \langle q_1 \mid t_1 \rangle \quad e \mapsto_Q \langle q_2 \mid t_2 \rangle}{\text{LET } v = a \text{ IN } e \mapsto_Q \langle q_1, \text{ LATERAL } q_2 \mid t_2 \rangle} \text{(LET)}$$

$$q_1 \equiv \frac{\begin{array}{l} a \mapsto_Q \langle q_1 \mid t_1 \rangle \quad e \mapsto_Q \langle q_2 \mid t_2 \rangle \\ \text{(WITH } t_1(v) \text{ AS MATERIALIZED} \\ \text{(SELECT } q_1 \text{ AS } t_1(v))) \text{ SELECT } * \text{ FROM } t_1(v) \end{array}}{\text{LET } v = a \text{ IN } e \mapsto_Q \langle q_1, \text{ LATERAL } q_2 \mid t_2 \rangle} \text{(LETMAT)}$$

$$\frac{\begin{array}{l} t \equiv \text{fresh row var} \quad e_1 \mapsto_Q \langle q_1 \mid t_1 \rangle \quad e_2 \mapsto_Q \langle q_2 \mid t_2 \rangle \\ b_1 \equiv \text{SELECT } t_1.* \text{ FROM } q_1 \text{ WHERE } v \\ b_2 \equiv \text{SELECT } t_2.* \text{ FROM } q_2 \text{ WHERE NOT } v \end{array}}{\text{IF } v \text{ THEN } e_1 \text{ ELSE } e_2 \mapsto_Q \langle (b_1 \text{ UNION ALL } b_2) \text{ AS } t \mid t \rangle} \text{(COND)}$$

$$q \equiv \frac{\begin{array}{l} t \equiv \text{fresh row var} \quad e_i \mapsto_Q \langle b_i \mid t_i \rangle_{i=1..n} \\ \text{(SELECT } t_1.* \text{ FROM } b_1 \text{ WHERE call='v}_1\text{' } \\ \text{UNION ALL} \\ \vdots \\ \text{UNION ALL} \\ \text{SELECT } t_n.* \text{ FROM } b_n \text{ WHERE call='v}_n\text{') AS } t \end{array}}{\text{CASE } v \text{ OF } \overline{v} : e \mapsto_Q \langle (q) \text{ AS } t \mid t \rangle} \text{(CASE)}$$

$$\frac{\begin{array}{l} t \equiv \text{fresh row var} \quad a \mapsto_Q \langle q_1 \mid t_1 \rangle \quad e \mapsto_Q \langle q_2 \mid t_2 \rangle \\ b_1 \equiv \text{SELECT false, NULL, v, true, NULL FROM } (q_1) \text{ AS } v \\ b_2 \equiv \text{SELECT } t_2.* \text{ FROM } q_2 \end{array}}{\text{EMIT } a; e \mapsto_Q \langle (b_1 \text{ UNION ALL } b_2) \text{ AS } t \mid t \rangle} \text{(EMIT)}$$

$$q \equiv \frac{\begin{array}{l} t \equiv \text{fresh row var} \\ q \equiv \text{apply substitution of the } n \text{ parameters in } \mathbb{Q} \end{array}}{\mathbb{Q}[v_1, \dots, v_n] \mapsto_Q \langle (\text{SELECT } q) \text{ AS } t \mid t \rangle} \text{(EMBED)}$$

Experiments 6.

The deliberately simple PL/SQL UDF `route` as introduced in Section 2.4 only uses a single embedded SQL query Q_1 . However, it did highlight the challenges associated with repeated `route` $\rightarrow Q_1$ context switching. The situation gets worse as we increase the complexity of the UDF. This chapter examines 18 PL/SQL UDFs, ranging in complexity from low to extremely high, and quantifies the runtime and memory impact that compiling to plain SQL can have on these UDFs. We also compare the performance of different PostgreSQL versions to see if the problems with PL/pgSQL have remained the same or not. We also measure if the compilation to SQL has improved between PostgreSQL versions. All of the following measurements have been performed on a 64-bit Linux x86 host (two AMD EPYC™ 7402 CPUs at 2.8 GHz and 2 TB of RAM). If not stated otherwise, the timings were averaged over five runs, ignoring the worst and best times.

6.1. Compiling a Collection of UDFs

Table 6.1 lists 18 UDFs that we use to measure the impact of the translation. Some of these UDFs are created based on the TPC-H benchmark, others are taken from the literature (mostly from the Froid paper), and some are our own creations. The UDFs implement a variety of algorithms—such as 2D/3D geometry routines, simulation of VMs and spreadsheets, or optimization problems over TPC-H data—on a multitude of built-in and user-defined data types (see column **Return Type**). We attempt to characterize the UDFs in terms of code size and loop nesting with columns **LOC** (lines of code), $|Q_i|$ (number of non-fast-path embedded SQL queries), and **Loop constructs**. Note that the number of loops determines the size of the ANF function family: `ray` consists of four loops (nested $\underline{\underline{\underline{\underline{\quad}}}}$ inside each other), resulting in the functions `start()`, $f_1()$, \dots , $f_4()$ after the ANF conversion. The UDFs `service` and `ship` contain branching control flow, but do not iterate at all. Their ANF conversion results in a single `start()` function, a case covered by Rule **NONREC** in Section 5.4. No trampoline or recursive CTE is generated. *Froid* [27] would output similar SQL code for these two UDFs. We will take a closer look at Froid in Chapter 7. **CC** (cyclomatic complexity) counts the number of independent control flow paths [69] through the PL/SQL UDFs and constitutes a measure of control flow complexity (LOC and CC for `route` are 22 and 3, respectively).

Compilation of UDFs from PL/pgSQL to SQL is based on the hypothesis that we can save context switching effort by working entirely on the SQL side of the fence. Column $Q \rightarrow f + f \rightarrow Q_i$ **overhead** quantifies this effort, which may even exceed the time invested in the actual UDF evaluation. For example, UDF `sight` wastes 69.8% of its runtime on overhead, instead of actual useful work. This overhead can be completely eliminated by compiling to pure SQL queries. This

[27]: Ramachandra et al. (2018), ‘Froid: Optimization of Imperative Programs in a Relational Database’

[69]: McCabe (1976), ‘A Complexity Measure’

Table 6.1.: A collection of PL/SQL UDFs with context switching overhead before and speedup after compilation to SQL.

UDF	Return type	Q _i	LOC	Loop constructs	CC	Q→f+f→Q _i overhead	Runtime (speedup) after compilation	Trampoline transitions	
<code>bbox</code>	detect bounding box of a 2D object	box	2	41	1 _u	5	32.5%	60.76% (1.65 ×)	1157
<code>force</code>	<i>n</i> -body simulation (Barnes-Hut quad tree)	point	3	43	1 _u	5	52.7%	51.72% (1.93 ×)	263
<code>global</code>	does TPC-H order ship intercontinentally?	boolean	1	20	1 _u	3	32.8%	69.60% (1.44 ×)	9
<code>items</code>	count items in hierarchy (adapted from [70])	int	2	27	1 _u	2	56.2%	32.97% (3.03 ×)	4097
<code>late</code>	find delayed orders (transcribed TPC-H Q21)	boolean	1	25	1 _u	4	34.3%	86.25% (1.16 ×)	9
<code>march</code>	track border of 2D object (Marching Squares)	point[]	2	40	1 _u	5	30.5%	72.41% (1.38 ×)	4568
<code>margin</code>	buy/sell TPC-H orders to maximize margin	row	3	57	1 _u	5	15.6%	83.68% (1.19 ×)	59
<code>markov</code>	Markov-chain based robot control	int	3	44	1 _u	3	15.9%	86.89% (1.15 ×)	1026
<code>packing</code>	pack TPC-H lineitems tightly into containers	int[][]	3	82	3 _u	9	45.6%	72.94% (1.37 ×)	312
<code>savings</code>	optimize supply chain of a TPC-H order	row	6	66	1 _u	4	42.0%	46.38% (2.16 ×)	9
<code>sched</code>	schedule production of TPC-H lineitems	row array	5	77	2 _u	6	31.8%	77.66% (1.29 ×)	33
<code>service</code>	determine service level (taken from [71])	text	1	22	0 _∅	3	28.1%	61.70% (1.62 ×)	0
<code>ship</code>	customer's preferred shipping mode	text	3	34	0 _∅	3	16.7%	76.73% (1.30 ×)	0
<code>sight</code>	compute polygon seen by point light source	polygon	3	49	2 _u	3	69.8%	69.13% (1.45 ×)	662
<code>visible</code>	derive visibility in a 3D hilly landscape	boolean	2	57	1 _u	3	42.8%	57.32% (1.74 ×)	258
<code>vm</code>	execute program on a simple virtual machine	numeric	1	28	1 _u	17	39.9%	77.46% (1.29 ×)	7166
<code>ray</code>	complete PL/SQL ray tracer (adapted from [72])	int[]	5	230	4 _u	25	22.6%	829.37% (0.12 ×)	59642
<code>sheet</code>	evaluate inter-dependent spreadsheet formulæ	float	9	117	4 _u	19	27.3%	186.33% (0.54 ×)	2811

The full definitions of the UDFs in Table 6.1 are given in Appendix A.

may improve performance beyond that: for example, PostgreSQL used 56.2% of the execution time for PL/pgSQL UDF `items` for context switching, but the function runs 3.03 times faster after compilation, see column **Runtime (speedup) after compilation**. This is exactly what the compilation is supposed to improve. The geometric mean speedup of all UDFs except `sheet` and `ray` is 51.45%.

However, compiling is not always advantageous. For example, UDFs `ray` and `sheet` show a slowdown after compilation. We included these UDFs primarily to demonstrate that the techniques of Chapter 3 scale to super-complex control flows. Such UDFs are rarely found in practice, and stretch the idea of computation close to the data. Performance suffers due to (1) the construction of large intermediate data structures, *e.g.*, arrays of 1800+ pixels in the case of `ray`, (2) extensive sets of live variables (44 for `ray`), and (3) complex control flow that leads to a high number of transitions through the trampoline (Column **Trampoline transitions**). While (1) and (2) affect the width, (3) determines the cardinality of the UNION table built by the recursive CTE. However, if we can keep the resulting memory pressure on the system's buffer cache down, things look considerably better even for these super-complex UDFs (see Section 6.3).

Reminder

```

1 u ← q1
2 w ← u
3
4 LOOP
5   i ← qm(w)
6   IF i = ∅ THEN BREAK
7   u ← u ∪ i
8   w ← i
9 RETURN u
```

Figure 6.1.: Semi-naive operational semantics for recursive CTEs.

In general, it is very important to minimize transitions through the trampoline. The semi-naive evaluation strategy used in `WITH RECURSIVE` queries (recall Section 3.1) requires copying the rows produced by the previous iteration to the working table of the next iteration. When `WITH RECURSIVE` needs to perform fewer iterations, this results in fewer rows appended to the resulting UNION table. This emphasizes the importance of optimizations that reduce the size of the ANF function family. Function inlining removes function calls that are passed through the `trampoline()`. For UDF `packing`, this optimization reduces the number of ANF functions from 6 to 3, which as a result reduces the transitions through `trampoline()` from 608 to 312. Also, the UNION table shrinks from 103 kB to 52 kB. Function inlining can also reduce the number of live variables, which in turn reduces the working table size.

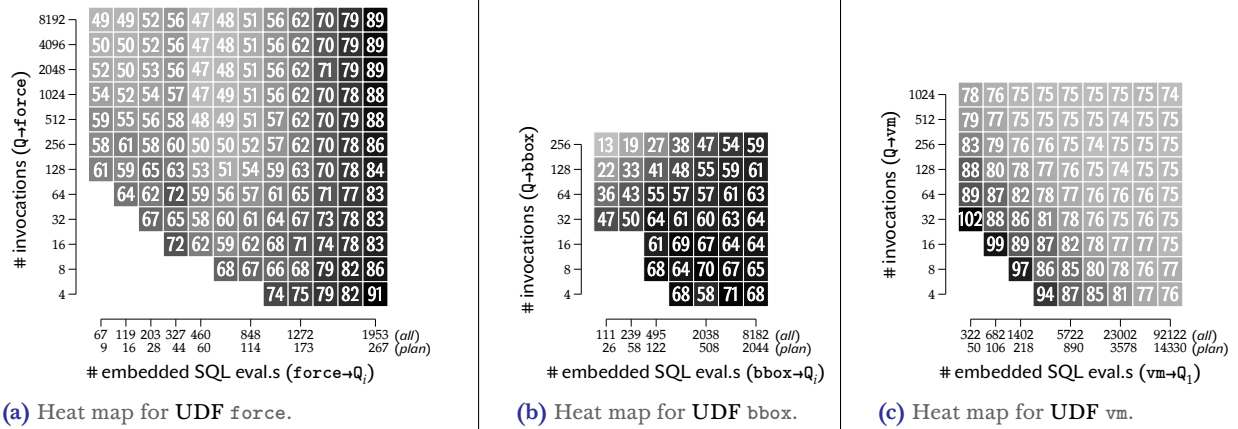


Figure 6.2.: Runtime (in % of PL/SQL UDFs) after compilation, $Q \rightarrow f$ and $f \rightarrow Q_i$ context switches varied (lower/lighter is better).

The Impact of $Q \rightarrow f$ and $f \rightarrow Q_i$. In the heat maps in Figure 6.2, we focus on the UDFs `force`, `bbox`, and `vm`. We vary the number of context switches ($Q \rightarrow f$: the number of invocations, $f \rightarrow Q_i$: evaluation of embedded SQL queries per invocation) during the experimental runs. We adapt the top-level query to control the number of UDF invocations. Similarly, we change the input size of the function to vary the number of intra-function iterations and thus the number of embedded SQL queries evaluated per invocation. This method provides some, but not exact, control over the total number of $f \rightarrow Q_i$ context switches and is the reason for the irregular $f \rightarrow Q_i$ axes in the heat maps. The value *all* refers to all such switches, while the value *plan* refers to the subset of those switches that required query planning and did not qualify for the fast path evaluation. The entries in the heat map show the runtime required after the compilation into SQL, relative to the original PL/SQL UDF. For example: 46 means that the compiled UDF ran in 46% of the original runtime, lower/lighter is better. Values less than 10 ms are ignored. For these UDFs, the runtimes per invocation range from (leftmost heat map column) 2.55 ms to 57.74 ms (rightmost column) after compilation.

When reading a heat map column from bottom to top, it is observed that as the number of UDF calls increases, the runtime of the compiled version decreases. This is because the $Q \rightarrow f$ overhead is avoided repeatedly. This phenomenon is particularly evident when the workload within f is minimal, which is the case on the left side of the heat maps. In this case, PL/SQL has limited ability to offset the costs associated with $Q \rightarrow f$ context switching.

The compiled UDF `force` (see Figure 6.2a) requires only 47% of the original runtime when many invocations are made. This is a significant improvement. As the size of the input increases, and with it the number of `trampoline()` iterations, this time increases to 89% of the PL/SQL runtime. To calculate `force`, the input size is based on the number of bodies that are represented by an underlying Barnes-Hut [73] quad tree. As the number of bodies increases, the tree depth and intermediate data structures also become larger. `force` uses a breadth-first traversal approach and stores any unexplored nodes in a SQL array. Both factors contribute to an increase in the

[73]: Barnes et al. (1986), ‘A Hierarchical $O(N \log N)$ Force-Calculation Algorithm’

size of the working table which in turn affects the runtime of the compiled function.

The original PL/SQL variant of `vm` performs a `vm`→`Q1` context switch for every instruction executed by the simulated VM. In the heat map for `vm`, the top-right run executes approximately 1024×14330 `vm`→`Qi` switches. The embedded query `Q1` is simple and executes quickly. Therefore, the expensive context switching from `vm` to `Q1` can never be compensated. This is a best-case scenario for compiling to SQL, with no context switches after compilation. The compiled version of `vm` executes approximately 25% faster in the top-right half of the heat map. As we approach the lower-left corner of the heat map and the number of opportunities for saving context switching decreases, the difference in runtime between the original and compiled versions becomes smaller.

6.2. PostgreSQL 11 vs. 15—What has changed?

[15]: Hirn et al. (2021), ‘One WITH RECURSIVE is Worth Many GOTOs’

1: The results have been validated by SIGMOD’s reproducibility efforts. All artifacts needed to repeat the experiments are publicly available here: <https://dl.acm.org/doi/abs/10.1145/3448016.3457272>

In [15], we first reported on the 18 UDFs in Table 6.1, measured with PostgreSQL 11.3¹, the most recent version at the time. There have been about four years of improvements between version 11.3 and version 15. The question is whether the problems with PL/SQL have remained the same or not. To answer this question, we performed a re-measurement of all UDFs with PostgreSQL 15.

We have found that PL/PgSQL exhibits nearly identical amounts of context switching overhead (recall column `Q`→`f`→`f`→`Qi` **overhead** in Table 6.1) for both PostgreSQL 11 and PostgreSQL 15. Hence, our main hypothesis remains valid. When comparing the runtime of our UDFs across the different PostgreSQL versions, it is evident that the system’s performance has improved. Figure 6.3 shows the speedup or, in some cases, slowdown values for both the PL/PgSQL version and the translations. On average, the execution time of all UDFs is 12.7% faster on PostgreSQL 15. The translations, on the other hand, improved by an average of only 4.6%. In comparison to their performance on PostgreSQL 11, some translations are even slightly slower.

This should not come as a surprise. Database systems are incredibly complex, and query optimization is particularly complicated. Identifying the best plan to execute can save several orders of magnitude in execution time compared to executing a bad plan. But the search space is huge, and an exhaustive search becomes impossible. For this reason, many databases use the classic SYSTEM R optimizer [74], which limits the search space to a manageable size. This is done using a cost model that estimates the processing time required and dynamic programming to eliminate duplicate work. However, the fact that such cost-based models are always only an approximation makes query optimization quite unstable [75, 76]. Even small changes to a system’s optimizer can have a big impact. Therefore, ensuring the reliability and robustness of database systems is critical so that they function correctly and efficiently. While these systems

[74]: Selinger et al. (1979), ‘Access Path Selection in a Relational Database Management System’

[75]: Haritsa (2010), ‘The Picasso Database Query Optimizer Visualizer’

[76]: Abhirama et al. (2010), ‘On the Stability of Plan Costs and the Costs of Plan Stability’

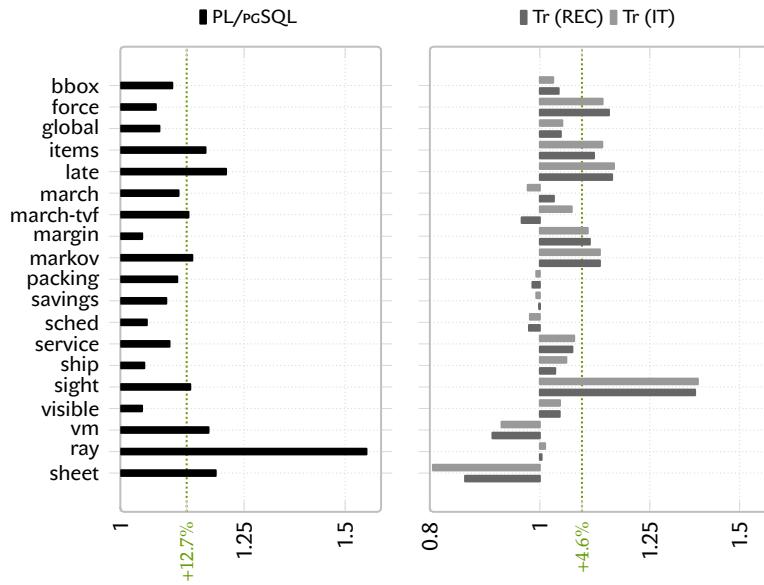


Figure 6.3. A runtime comparison of PostgreSQL 11 and PostgreSQL 15 for both, PL/pgSQL, and the corresponding translations. The dotted line (.....) shows the geometric mean of the speedup values.

are always trying to improve performance without performance regressions in the general case, some changes can end up degrading performance for some queries. This is especially true for complex queries that are difficult to optimize. In such cases, the optimizer may choose a suboptimal plan. This is exactly what happens to some of our UDF translations. The PostgreSQL 15 optimizer chooses a different plan than the PostgreSQL 11 optimizer, which is slower for some of our examples. This is not a problem with the translation, but rather a consequence of the complexity of the system.

Simple Expressions. The performance of the PL/SQL version of `ray` improved by approximately 56%. This is more than twice the speedup of any other UDF. So we took a closer look. To do so, we used a *stack trace visualizer*². This tool is used to display hierarchical data as a *flame graph* and can help to identify the most frequent code paths in programs. Comparing the two flame graphs of both versions of PostgreSQL shows a difference in the PL/pgSQL function `exec_eval_expr`. This function is used to execute embedded SQL queries and expressions. It also performs fast path evaluation for simple expressions as explained in Section 2.3.1. While there are just a few queries in `ray`, there are many simple expressions that PL/SQL evaluates using the fast path. It turns out, that this fast path has been improved in a recent version of PostgreSQL. We were able to pinpoint exactly which patch was responsible for this speedup³. This patch has been part of PostgreSQL since version 13⁴ and optimizes the plan caching PL/pgSQL uses for simple expressions. According to the patch notes, the overhead of managing plans can be greater than the cost of evaluating simple expressions. Additionally, the patch is said to improve performance by about 2 times. Although not all the speedup is directly due to this patch, it still explains more than half of it. The rest of the improvement can be attributed to a general enhancement of the system’s performance. `ray` is a best-case scenario for PL/SQL, but a worst-case scenario for our compilation strategy. This is because the function contains many

Table 6.2. A runtime comparison of PostgreSQL 11 and PostgreSQL 15.

UDF	Speedup (%)		
	PL/pgSQL	REC	IT
bbox	9.81	3.89	2.75
force	6.60	15.05	13.58
global	7.30	4.35	4.67
items	16.59	11.61	13.50
late	20.99	15.79	16.23
march	11.06	2.90	-2.44
march-tvf	13.07	-3.62	6.68
margin	4.01	10.68	10.21
markov	13.85	12.96	12.96
packing	10.81	-1.49	-0.72
savings	8.63	-0.13	-0.72
sched	4.89	-2.18	-2.00
service	9.27	6.82	7.22
ship	4.41	3.17	5.53
sight	13.43	37.02	37.79
visible	3.99	4.12	4.18
vm	17.23	-9.16	-7.48
ray	55.84	0.36	1.08
sheet	18.75	-14.07	-19.50

2: <https://github.com/brendangregg/FlameGraph>

3: Patch: <https://github.com/postgres/postgres/commit/8f59f6b>



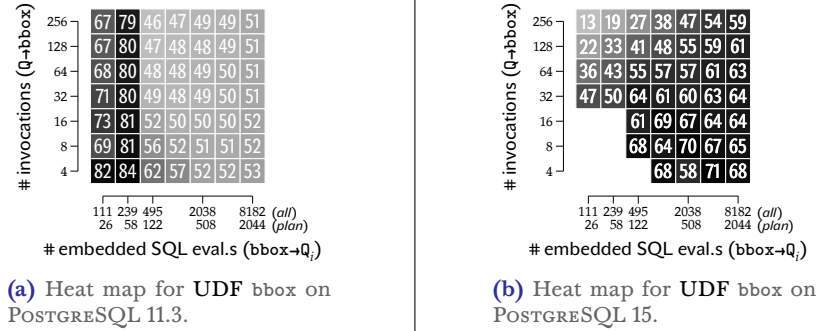
Discussion: <https://www.postgresql.org/message-id/CAFj8pRDRVfLdAxsWeVLzCAbkLFZhW549K+67tp0c-faC8uH8zw@mail.gmail.com>



4: <https://www.postgresql.org/docs/release/13.0/>

“Improve performance of simple PL/pgSQL expressions (Tom Lane, Amit Langote)”

Figure 6.4.: Runtime of UDF `bbox` (in % of PL/SQL UDFs) after compilation on PostgreSQL 11.3 and PostgreSQL 15, $Q \rightarrow f$ and $f \rightarrow Q_i$; context switches varied (lower/lighter is better).



simple expressions executed in deeply nested loops, but very few queries. This results in only 22.6 % context switching overhead. The evaluation of recursive SQL queries is not free. As a result, the evaluation overhead is greater than the PL/SQL overhead, which leads to bad performance of the translation.

6.2.1. Cost-based Optimization

Figure 6.4 shows two heat maps for UDF `bbox`. The heat map on the left is measured on PostgreSQL 11, and the one on the right is measured on PostgreSQL 15. On PostgreSQL 11, there is a notable dark-light boundary at 239 `bbox \rightarrow Q_i` embedded SQL query evaluations in the heat map for UDF `bbox`. This is due to a change in the query plan. PostgreSQL 11 switches from a `SEQ SCAN` to an `INDEX SCAN` for table `squares` as the input table size increases (see Figure 6.5), resulting in a significant runtime reduction for the compiled function. The situation is different for PostgreSQL 15. The query plan does not change in this version of PostgreSQL, and the performance on the left side of the heat map (Figure 6.4b) is exceptionally good—we will get to this in the next paragraph. After compilation, the two SQL queries embedded in the `bbox` are planned and optimized as a whole. Furthermore, the optimizer knows the sizes of the input tables and can use this information for planning. Runtime improvements beyond the saved context switching overhead are due to such multi-query optimization effects [77]. Multi-query optimizer decisions do not apply to PL/SQL UDFs. PL/SQL is unable to merge multiple statements into one. Such an optimization would be required in order to combine multiple SQL queries into one. Thus, the SQL optimizer and executor only ever receive one SQL query at a time, which prevents multi-query optimizations. This is a fundamental limitation of PL/SQL. The only way to overcome this limitation is to compile the UDF into a single SQL query. This allows the optimizer to plan the query as a whole. This is exactly what our compilation strategy is designed to do.

[77]: Sellis (1998), ‘Multiple-Query Optimization’

Memoization. PostgreSQL 14 introduced memoization plans that allow the system to cache results from parameterized scans within nested-loop joins. This type of plan allows skipping scans to the underlying plans when the current parameter results are already cached. This is implemented by the `MEMOIZE` operator. Memoization

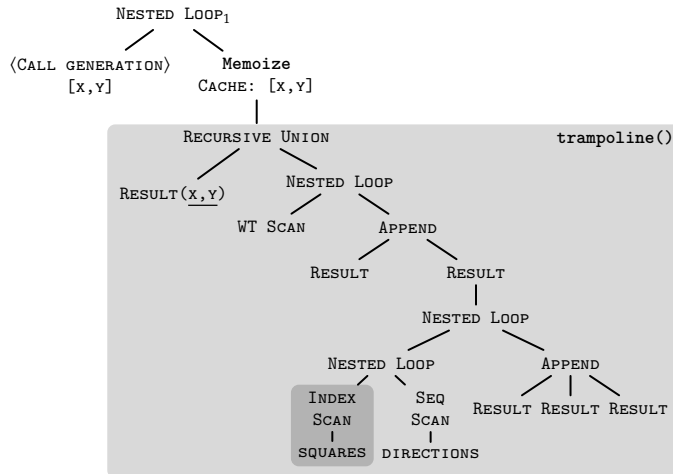


Figure 6.5.: Slightly simplified physical plan of `bbox` on PostgreSQL 15.

can reduce runtime significantly in the presence of duplicates on the left side (*i.e.*, the binding or call generating side) of the `NESTED LOOP` join by eliminating duplicate work. The physical plan of UDF `bbox` on PostgreSQL 15 makes use of this feature (see Figure 6.5). This plan utilizes memoization for the top-level `NESTED LOOP1` join, which generates parameters `x` and `y` for each invocation. These invocations are not unique, meaning that the UDF can be called multiple times with the same parameters. During execution of this plan, the `MEMOIZE` operator first checks whether a cached result already exists for the pair of parameters `[x,y]`. The `RECURSIVE UNION` operator, which implements the `trampoline()`, is only executed if there is no cached result. The trampoline execution is the most computationally intensive task, so the reduction of duplicate work can be a significant time saver.

The probability of these duplicates is highest on the left side of the heat map and decreases towards the right side. On the left side of the heat map, almost all invocation results are cached. This effect decreases as the number of duplicate calls reduces.

6.3. To Recurse is Divine, to ITERATE Space-Saving

The evaluation of the recursive CTE that forms the SQL trampoline builds a `UNION` table `run` that contains a trace of all trampoline transitions. While this stack-like trace can be revealing and aid in function debugging [14], its construction can consume significant memory. This is especially true for compiled functions that maintain many live variables with large contents (resulting in *wide* `UNION` tables) and transition through the trampoline often (*long* `UNION` tables containing one row per transition). We have recorded the size of the resulting `UNION` table in Table 6.3 under the heading `WITH RECURSIVE`. We have excluded the non-looping UDFs `service` and `ship` as they do not allocate a `UNION` table at all. Table 6.3 reports worst-case runs in which the functions processed large inputs, corresponding to the rightmost columns in the heat maps in Figure 6.2.

[14]: Hirn et al. (2020), ‘PL/SQL Without the PL’

POSTGRESQL stores the entire `UNION` table in secondary memory if it exceeds the buffer space allocated for the `WITH RECURSIVE` evaluation. Table 6.3 also shows the number of page writes for buffer sizes of 512 kB and 100 MB. UDFs with a large number of trampoline transitions (`ray`, `vm`) and/or significant live variable sets (`march`, `ray`, `sheet`) are particularly noticeable.

This can be countered by noting that all rows except those with `"rec?" = false` or `"data?" = true` are discarded from the `UNION` table before the final function result is returned. The control rows of the stack-like trace of the computation are never consumed and never need to be stored.

We built this behavior directly into POSTGRESQL in the form of `WITH ITERATE`, a variant of `WITH RECURSIVE` that stores only rows where the `"data?"` column is set to `true`. Note that compared to [13–15], the version described here is a slightly generalized version of `WITH ITERATE`. Instead of ignoring all but the last row in the working table, all data rows are collected. This generalized semantics is necessary for appropriate handling of table-valued functions. Effectively, the predicate `"data?"` is pushed down into the computation of the recursive CTE—otherwise, its semantics remain as is. In a VOLCANO-style query execution engine [78], `WITH ITERATE` does not return to its parent operator before producing a data row. The space savings can be significant, although the query evaluation time is only marginally reduced. We only ever allocate space for data rows. For scalar UDFs, that is exactly one row, so no page I/O is required regardless of the buffer size (see columns under heading `WITH ITERATE` in Table 6.3). It is precisely those compiled UDFs that have been penalized the most by the `UNION` table construction that benefit from this. A one-line change from `WITH RECURSIVE` to `WITH ITERATE` in the SQL trampoline in Rule REC restores the low memory requirements of the original PL/SQL UDF. The complex UDF `sheet` is sped up by a factor of 2.21×. UDFs `ray` and `march` benefit less, but the speedups are still significant.

`WITH ITERATE` can provide significant runtime and space savings. But, its implementation reaches into the POSTGRESQL kernel, defying the ideal of a pure source-to-source compilation approach. So we consider `WITH ITERATE` to be an optimization rather than an integral part of the core design.

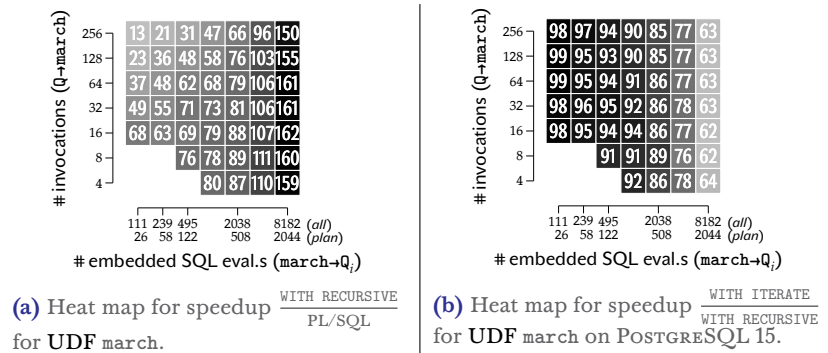
[13]: Duta et al. (2020), ‘Compiling PL/SQL Away’

[14]: Hirn et al. (2020), ‘PL/SQL Without the PL’

[15]: Hirn et al. (2021), ‘One WITH RECURSIVE is Worth Many GOTOs’

[78]: Graefe (1994), ‘Volcano—An Extensible and Parallel Query Evaluation System’

Figure 6.6.: Runtime of UDF `march` after compilation. $Q \rightarrow f$ and $f \rightarrow Q_i$ context switches varied (lower/lighter is better).



UDF	WITH RECURSIVE			WITH ITERATE		Speedup over RECURSIVE	
	UNION table	# writes 512 kB	100 MB	working table	# writes		
bbox	122 kB	55 793	0	161 b	0	marginal	
force	1155 kB	633 756	0	81 b	0		
global	789 b	0	0	85 b	0		
items	448 kB	100	0	69 b	0		
late	10 kB	0	0	1192 b	0		
margin	7 kB	0	0	121 b	0		
markov	40 kB	0	0	41 b	0		
packing	52 kB	0	0	580 b	0		
savings	2 kB	0	0	714 b	0		
sched	21 kB	0	0	1088 b	0		
sight	3544 kB	14 143	0	15 kB	0		
visible	40 kB	0	0	152 b	0		
vm	1643 kB	198 849	0	432 b	0		
march	256 MB	33.5 M	33.5 M	256 kB	0		1.54×
ray	244 MB	2 M	2 M	15 kB	0		1.32×
sheet	100 MB	6.5 M	6.5 M	62 kB	0		2.21×

Table 6.3.: WITH RECURSIVE vs. WITH ITERATE: Memory allocation and buffer page writes due to UNION table construction.

Related Work and Conclusion

7.

7.1. Conclusions

With PL/SQL, we can move computations to where the data lives. This avoids network latency and unnecessary data transfers in general. The PL/SQL interpreter has direct access to table data and sits right inside the database. Still, the tension between imperative and declarative programming and their respective execution modes spoils the promising PL/SQL idea [13–15, 27]. However, the need for complex in-database computation is only going to grow. This is exemplified by current efforts to host machine learning algorithms within database systems [79–82]. While there are efforts to express such algorithms in regular programming languages and compile them into SQL queries [83, 84], we believe that PL/SQL is particularly worthy of study. PL/SQL’s wide availability, its SQL-native type system, and its seamless embedding of SQL queries make it particularly well-suited for data-oriented computation.

While the performance of such PL/SQL functions is often disappointing, we were able to prove that our compilation method can significantly improve the situation. The experiments in Chapter 6 show that for 18 UDFs, the runtime after compilation improves by an average of 51.45%. However, not all UDFs benefit from compilation. Functions that collect large intermediate data structures or that do not have much context switching overhead due to embedded SQL queries can be slowed down. It is not possible to statically determine whether a UDF benefits from compilation or not. But these findings can be used as a rule of thumb.

The inference rules in Chapter 5 are tailored to compile PL/SQL functions into SQL, but the general idea is more universal. The realization that trampolined-style conversion can be used to express any computation in SQL is an important contribution of this work, and can in principle be used to compile any imperative control flow into SQL. The BYEPY project (see Section 7.4) has shown that we can attach a variety of different frontends to this set of rules, as long as the language can be compiled into our SSA IR. The same is true for different backends. While the last rule set generates the SQL dialect of PostgreSQL, any database system that supports LATERAL joins and recursive CTEs can be targeted. This allows imperative PL/SQL execution on database systems without the need to implement a PL/SQL interpreter—which may be impossible to do in case of closed-source systems—and without the drawbacks associated with that.

Functional SSA. Our PL/SQL to SQL compilation (recall Chapters 3 and 5) relies on the imperative SSA IR with ϕ functions, and the functional ANF IR. In this first part of the thesis, ANF is mainly

[13]: Duta et al. (2020), ‘Compiling PL/SQL Away’

[14]: Hirn et al. (2020), ‘PL/SQL Without the PL’

[15]: Hirn et al. (2021), ‘One WITH RECURSIVE is Worth Many GOTOs’

[27]: Ramachandra et al. (2018), ‘Froid: Optimization of Imperative Programs in a Relational Database’

[79]: Boehm et al. (2019), *Data Management in Machine Learning Systems*

[80]: Jankov et al. (2019), ‘Declarative Recursive Computation on an RDBMS (or Why You Should Use a Database for Distributed Machine Learning)’

[81]: Feng et al. (2012), ‘Towards a Unified Architecture for in-RDBMS Analytics’

[82]: Blacher et al. (2022), ‘Machine learning, linear algebra, and more: Is SQL all you need?’

[83]: Emani et al. (2016), ‘Extracting Equivalent SQL from Imperative Code in Database Applications’

[84]: Grust et al. (2010), ‘Avalanche-Safe LINQ Compilation’

[85]: Lattner et al. (2020), ‘MLIR: A Compiler Infrastructure for the End of Moore’s Law’

[9]: Chakravarty et al. (2004), ‘A functional perspective on SSA optimisation algorithms’

[57]: Appel (1998), ‘SSA is Functional Programming’

[86]: Wang et al. (2020), ‘RASQL: A Powerful Language and Its System for Big Data Applications’

[87]: Passing et al. (2017), ‘SQL-and Operator-centric Data Analytics in Relational Main-Memory Databases.’

[88]: Neumann et al. (2020), ‘Umbra: A Disk-Based System with In-Memory Performance.’

[89]: Sichert et al. (2022), ‘User-defined operators: Efficiently integrating custom algorithms into modern databases’

[16]: Hirn et al. (2023), ‘A Fix for the Fixation on Fixpoints’

used to determine the columns of the working table, and to establish TRAMPOLINED STYLE. Both operations would be possible using an alternative to this design that does not require the use of ANF. MLIR [85] uses a *functional SSA* IR, that combines the properties of SSA and ANF. Instead of using ϕ functions, MLIR uses a functional form of SSA in which basic blocks are parameterized and **GOTO** statements explicitly pass values to these block arguments defined by the successor block. This effectively represents the same information as using the combination of SSA and ANF, but with a single IR. While fewer IRs are generally desirable as it reduces the number of data structures, in the context of this thesis it makes sense to keep the ANF IR. This is because we will be reusing parts of the PL/SQL to SQL rule set in the following Part ‘Functional Programming on Top of SQL Engines’. Also, the compilation effort would not be reduced by using this alternative design. To translate from SSA to ANF, we instantiated a customized version of Chakravarty, Keller, and Zadarnowski’s algorithm. This algorithm uses the information encoded by SSA’s ϕ functions to remove all basic blocks and replace them with a mutually recursive function family. The *exact same* information is needed to create the MLIR-style functional SSA IR. Therefore, the choice between using a classical SSA in combination with ANF, or using a functional SSA IR boils down to a mere design decision. While there may be local advantages to using one or the other, overall the same algorithms are required because the same transformations must be applied. This is not surprising, since the correspondence between SSA and ANF has been shown many times [9, 57].

Alternative Trampoline drivers. Recursive CTEs are great and irreplaceable for this research, but they often suffer from suboptimal performance. Efficient execution strategies beyond semi-naive evaluation is often neglected by database systems. This is because the feature is not widely understood and is therefore sometimes perceived as *alien* to the language. This limits the number of users of the feature, and thus the pressure on database providers to improve it. While most major database systems support recursive CTEs, there has not been much research into more efficient execution strategies. While the outcome of such a research effort is unclear, there are certainly cases that can be optimized [86, 87]. Although our compilation strategy produces recursive CTEs that theoretically require only a single **WHILE** loop, even modern code-generating database systems such as Umbra [88] cannot completely eliminate the overhead of fixed-point evaluation [89].

Trampolined style and its characteristic limited complexity of control flow may allow for different execution strategies. In [16], we explored several alternative semantics for recursive CTEs that differ from the fixed-point semantics. Similar to this branch of research, a “**WITH TRAMPOLINE**” database operator, specifically designed to execute trampolined-style SQL, could improve performance and/or memory usage. Without the need to adhere to the fixed-point semantics of recursive CTEs, such an operator could employ different parallelization strategies.

7.2. Related Work: Froid

FROID [27, 34] is described as an extensible framework for optimizing imperative programs in relational database systems. The basic idea is to transform UDFs into a SQL-compatible form so that the UDF can be embedded into the calling SQL query. Similar to our approach, FROID also converts sequences of PL/SQL statements into subqueries. These subqueries are concatenated using SQL SERVER’s `OUTER APPLY`. This similarity to our approach is no coincidence, since we have adapted and generalized FROID’s statement chaining technique to compile loop-free control flow structures. FROID’s approach is elegant and straightforward, but it has serious limitations: in particular, the translation only works for functions that have a loop-free control flow. This limitation does not diminish its value for loop-free UDFs, but it does limit the applicability of FROID.

Translation Rules. This paragraph is partially based on [90]. FROID performs its translation on the basis of relational algebra operators. It is not implemented and described as a SQL-to-SQL translation. This may be the reason why there is no formal set of translation rules in the relevant publications. In the following, we present an approximate set of formal translation rules, reverse-engineered from the description in the publications, and from the actual implementation in SQL SERVER. This was done by comparing the execution plans generated by FROID with the execution plans of translations using the inference rules. Note, however, that this set of rules may not always perfectly match FROID’s output. In particular, the actual implementation may perform additional optimizations that are not described in the publications. Nevertheless, the rules provide a useful basis for learning how FROID’s translation works, and can serve as a valuable resource for understanding its optimization.

Target input UDFs must conform to the grammar shown in Figure 7.1. These UDFs are loop-free, but can include statement sequences and branching control flow. The translation \Rightarrow_f assumes that *all* control flow paths of the input UDF end in `RETURN a` and return a result value. As a result, branching control flow paths never merge, which allows for a compact and simple set of rules that avoids the use of SSA. This normalization of the UDFs may result in some duplication of code, but it preserves the length of the control flow path (no additional work is done at runtime). The rules map statements s to the pair $\langle q \text{ AS } t(c) \mid t \rangle$, where the scalar SQL query q returns its result in column $t.c$ (the row variable t is made explicit to facilitate translation of statement sequences). Assignment statements are handled by Rule `ASSIGN`. Column $c \equiv v$ for the assignment `SET v = a`. Otherwise $c \equiv \text{retVal}$, *i.e.*, the statement’s result value (Rules `IF`, `IF-ELSE` and `RETURN`). Sequences of statements $s_0 s_1$ are translated as `LATERAL` joins (Rule `SEQ`). This join type allows the query q_1 corresponding to s_1 to access the values of variables bound by statement s_0 ’s associated query q_0 through its row variable t_0 . Sequences of n statements lead to a chain of $n - 1$ `LATERAL` joins. This is similar to our translation as described in Section 5.4.

[27]: Ramachandra et al. (2018), ‘Froid: Optimization of Imperative Programs in a Relational Database’

[34]: Ramachandra et al. (2019), ‘BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid’

[90]: Franz et al. (2024), ‘Dear User-Defined Functions, Inlining isn’t working out so great for us. Let’s try batching to make our relationship work. Sincerely, SQL’

```

f ::= CREATE FUNCTION v(v τ, ..., v τ)
    RETURNS τ AS BEGIN d; s END;
d ::= DECLARE v τ
    | d; d
s ::= SET v = a;
    | IF a s [ELSE s] ENDIF;
    | RETURN a;
    | BEGIN s END
    | s s
a ::= scalar SQL expression
v ::= <UDF/variable identifier>
τ ::= <scalar SQL type>

```

Figure 7.1.: FROID SQL UDF Grammar.

$$\begin{array}{c}
\frac{s \Rightarrow_i \langle q|t \rangle}{\text{CREATE FUNCTION } v(v_0 \ \tau_0, \dots, v_n \ \tau_n) \ \Rightarrow_i \ \text{SELECT } t.\text{retVal} \\ \text{RETURNS } \tau_r \ \text{AS BEGIN } d; \ \text{s END}; \ \text{FROM } q} \text{(INLINE)} \\
\\
\frac{s_0 \Rightarrow_i \langle q_0|t_0 \rangle \quad s_1 \Rightarrow_i \langle q_1|t_1 \rangle}{s_0 \ s_1 \Rightarrow_i \langle q_0, \text{LATERAL } q_1|t_1 \rangle} \text{(SEQ)} \quad \frac{s \Rightarrow_i \langle q|t \rangle}{\text{BEGIN } s \ \text{END } \Rightarrow_i \langle q|t \rangle} \text{(BLOCK)} \\
\\
\frac{s_0 \Rightarrow_i \langle q_0|t_0 \rangle \quad t \equiv \text{fresh row var}}{\text{IF } a \ s_0 \ \text{ENDIF}; \Rightarrow_i \left\langle \begin{array}{l} \text{(SELECT CASE WHEN } a \\ \text{THEN (SELECT } t_0.\text{retVal FROM } q_0) } \\ \text{END) AS } t(\text{retVal}) \end{array} \right\rangle} \text{(IF)} \\
\\
\frac{s_0 \Rightarrow_i \langle q_0|t_0 \rangle \quad s_1 \Rightarrow_i \langle q_1|t_1 \rangle \quad t \equiv \text{fresh row var}}{\text{IF } a \ s_0 \ \text{ELSE } s_1 \ \text{ENDIF}; \Rightarrow_i \left\langle \begin{array}{l} \text{(SELECT CASE WHEN } a \\ \text{THEN (SELECT } t_0.\text{retVal FROM } q_0) } \\ \text{ELSE (SELECT } t_1.\text{retVal FROM } q_1) } \\ \text{END) AS } t(\text{retVal}) \end{array} \right\rangle} \text{(IF-ELSE)} \\
\\
\frac{t \equiv \text{fresh row var}}{\text{SET } v = a; \Rightarrow_i \langle \text{(SELECT } a) \ \text{AS } t(v)|t \rangle} \text{(ASSIGN)} \\
\\
\frac{t \equiv \text{fresh row var}}{\text{RETURN } a; \Rightarrow_i \langle \text{(SELECT } a) \ \text{AS } t(\text{retVal})|t \rangle} \text{(RETURN)}
\end{array}$$

Figure 7.2.: Defines the UDF translation from PL/SQL to plain SQL, following the FROID-style UDF inlining strategy.

[24]: Gupta et al. (2020), ‘Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates’

[71]: Simhadri et al. (2014), ‘Decorrelation of User Defined Functions in Queries’

7.3. Related Work: Aggify

With AGGIFY [24], the FROID team has developed an approach that can include limited forms of looping. AGGIFY compiles loops into user-defined aggregate functions and places the loop logic inside the aggregate’s accumulator routine. This approach was first proposed in [71]. AGGIFY manages to improve performance significantly. For some use cases, the runtime is reduced by several orders of magnitude. The authors also propose a two-step approach in which a UDF is first loop-eliminated using AGGIFY, and then the now loop-free UDF is eliminated using Froid. The authors call this scenario AGGIFY+.

However, there are some drawbacks to this approach. Aggregate functions, by their nature, cannot be aborted prematurely, they generally read their entire input. This inflexibility results in the necessity that the number of loop iterations must be known in advance. It also means that a conditional early EXIT, for example, cannot stop the iteration. If the number of iterations is high, this can be problematic, and the loop may exit prematurely, resulting in poor performance. Another notable drawback of AGGIFY lies in the proposal to implement the accumulator function in C#. While this choice makes sense in some contexts, it is problematic for approaches that target plain SQL. We believe that the use of trampoline-based PL/SQL compilation, which allows *completely arbitrary* control flow, is a clear advantage in this case.

Despite the drawbacks of implementing the main logic of the ag-

gregate function in C#, AGGIFY manages to significantly improve the performance of iterative UDFs in SQL SERVER. They were able to show that their approach can reduce runtime by several orders of magnitude for some examples. They also used AGGIFY to move database-external computations implemented in JAVA into the database system. The result was a reduction in the runtime and the amount of data transferred.

7.4. Related Work: ByePy

As part of the BYEPY project [91, 92], Fischer applied the compilation chain as described in Chapter 5 to PYTHON functions. The focus was on data-intensive PYTHON functions using PSYCOPI2, a widely used database API targeting PostgreSQL backends. Such functions suffer from many of the same problems as PL/SQL functions and external database applications, *i. e.*, a constant back and forth between the PYTHON interpreter and the SQL database engine. This is problematic because the context switching overhead for such PYTHON functions is drastically higher than that of PL/SQL UDFs, since PYTHON is typically executed externally and not right inside the database kernel. Such programs suffer from both a structural and a conceptual impedance mismatch. This results in a lot of friction during execution. Fischer was able to reduce run times by up to two orders of magnitude by compiling these PYTHON programs into a pure SQL query. Note that these measurements were taken without network latency. The PYTHON programs and the database system were hosted on the same machine. Introducing network latency makes the situation worse.

Instead of using PL/SQL UDFs as input to the first compilation stage, Fischer developed a novel PYTHON front-end, for a restricted subset of the language. No changes have been made to the backend. BYEPY exemplifies the flexibility and adaptability of our compilation process, even when it is applied to different programming languages and environments. The BYEPY project shows that the compilation process can be used to improve the performance of database-external programs. This is a promising result, as it shows that the compilation process is not limited to PL/SQL.

7.5. Related Work: User-Defined Operators

Sichert and Neumann present User-defined operators (UDOs) [89] as a concept for moving complex algorithms inside database systems. Their system allows users to write code in any programming language. This code can then be compiled and integrated into the existing database system. Instead of using SQL to express these algorithms, UDOs use algebraic operators to extend the relational algebra used by database systems. This allows users to write algorithms in a more natural way, without having to worry about the limitations of SQL. UDOs can handle arbitrary control flow, loops, and conditionals. Table-valued functions are also valid input.

[91]: Fischer et al. (2022), ‘Snakes on a plan: Compiling python functions into plain SQL queries’

[92]: Fischer (2023), ‘To Iterate Is Human, to Recurse Is Divine—Mapping Iterative Python to Recursive SQL’

[89]: Sichert et al. (2022), ‘User-defined operators: Efficiently integrating custom algorithms into modern databases’

[88]: Neumann et al. (2020), ‘Umbra: A Disk-Based System with In-Memory Performance.’

1: <https://github.com/tum-db/user-defined-operators>

UDOs are currently implemented for both UMBRA [88] and PostgreSQL¹. In the case of UMBRA, the UDO query compiler takes the user input and integrates it into an existing query plan. Interfacing with database system internals is handled by the UDO compiler, so the user does not need to know about them. UMBRA is a compiling database system, which means that query execution is not done in the classical interpreted way. The system uses its own LLVM IR for this. The UDO compiler also compiles into this IR, which allows further optimization of the user-provided code. Since PostgreSQL does not compile queries, these optimizations are not possible in this case.

While UDOs support similar features as input as we do, modifications to the database internals are required to make UDOs work. This limits the applicability of the approach, as not all systems support UDOs or can be modified to do so. PostgreSQL is open source, so a custom version can be created and distributed, but unless the functionality is merged into the code base, users will have to compile the system themselves.

UDOs do not have to rely on recursive CTEs to emulate loops and arbitrary control flow in general. Since this SQL construct comes with its own overhead, this can be beneficial for performance. However, compared to a SQL solution, the UDF code remains separate unless it is optimized at the LLVM IR level. Such an optimization is only possible for code-generating systems.

7.6. Related Work: Tupleware

[93]: Crotty et al. (2015), ‘Tupleware: Big Data, Big Analytics, Small Clusters.’

[94]: Crotty et al. (2015), ‘An Architecture for Compiling UDF-Centric Workflows’

TUPLEWARE [93] is a system optimized for compute-intensive, in-memory analytics on small clusters. The system combines ideas from the database and compiler communities to create an easy-to-use and highly efficient end-to-end data analytics solution. Recognizing the growing need for custom algorithms expressed as workflows of UDFs [94], TUPLEWARE provides a language-agnostic solution that allows users to use their preferred programming language by compiling the language into LLVM code. This allows the system to collect statistics for low-level optimizations.

This architecture allows UDF-centric workflows to be compiled using data statistics, UDF properties, and the underlying hardware to optimize the generated code. Crotty et al. have shown that the TUPLEWARE prototype can achieve orders of magnitude speedups over alternative systems.

The idea is related to UDOs as both approaches use LLVM to compile into a common IR. TUPLEWARE also recognizes the importance of UDFs and the need for efficient execution, so the system compiles to LLVM code, which makes sense for code-generating systems. However, the approach still suffers from a lack of portability. Without support from the database system, the compilation cannot be used. While this is not necessarily a problem for newly designed database systems, legacy systems most likely cannot use these ideas.

7.7. More Related Work

Blacher et al. [82] also recognize the Turing completeness of SQL, and that the language is capable of expressing arbitrary computations. In their approach, they use `UNION ALL` and `WITH RECURSIVE` clauses to express basic imperative primitives in SQL to write complex algorithms using SQL. Their approach is closely related to the method described in this thesis, but differs in details. While they also use `WITH RECURSIVE` to express looping control flow, they choose a different encoding for variables. Where we use `LATERAL` joins, they use `WITH` clauses instead. Conditionals, on the other hand, are encoded exactly the same. Similar to the BYEPY project, they compile PYTHON programs. Since these programs can fail, they added error support. They applied this approach to several machine learning algorithms and compared the performance of NUMPY, HYPER [95], and POSTGRESQL. Although NUMPY is a dedicated high-performance computing package for PYTHON, Blacher et al. showed that the SQL version executed by HYPER can outperform NUMPY by a factor of 3. This supports our hypothesis and shows that SQL-only algorithms are not a gimmick, but have high practical value.

[82]: Blacher et al. (2022), ‘Machine learning, linear algebra, and more: Is SQL all you need’

[95]: Kemper et al. (2011), ‘HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots’

FUNCTIONAL PROGRAMMING ON TOP OF SQL ENGINES

Recursive SQL UDFs

8.

In the first part of this thesis, we described a compilation method that is capable of compiling *imperative* PL/SQL UDFs into plain SQL. We have developed formal sets of rules that allow us to compile such UDFs with arbitrary control flow structures, *i. e.*, (deeply nested) loops, conditionals, variable assignments, *etc.*, and express them in a SQL-compatible form. However, this compilation method can not handle any form of *recursive* UDFs.

In this part of the thesis, we will shift the focus to recursive functions and show that the compilation method can be extended to handle recursion as well. We will extend and modify the compilation chain as needed to transform recursive UDFs into a SQL-compatible form, and demonstrate the effectiveness of the compilation through an experimental study. To do this, we will again use long-established ideas and techniques from the programming language community, namely continuation passing style (CPS) [96] and defunctionalization [19]. Using these two techniques, any recursive program can be transformed into a first-order, tail-recursive program with an explicit stack [97–100]. In effect, we turn the program into a state machine [101]. This way, we can once again use trampolined style to compile the UDF into a single recursive SQL query.

8.1. From 1000s of Plans to *One* Plan

SQL database engines are experts at plan-based query execution. Engine internals are specifically designed to support query-to-plan compilation, optimization through plan rewriting, and the—often interpreted—evaluation of the resulting plans. This idea works well for SQL and has been proven to be powerful many times over the years. However, SQL has its limitations, and developers often need more expressive or ergonomic language features. One manifestation of such a language feature, or extension, are SQL scripting languages such as PL/SQL.

Some database systems support `LANGUAGE SQL` UDFs, also known as *query language functions*. These functions do not support imperative language features. They simply execute an arbitrary list of SQL statements and return the result of the last query in the list. SQL UDFs also receive plan-centric treatment, but the results can only be described as disappointing. UDF runtime performance is often subpar, and it is a well-known fact in the SQL development community that UDFs should be avoided as a result [15, 27, 102]. In fact, SQL applications incur a cost with every function call due to the engines' plan-based approach to UDF evaluation.

To illustrate this, consider the UDF `floyd(n, s, e)` in Figure 8.1, which implements *Floyd & Warshall's* algorithm [103] for finding the length of the shortest path between nodes `s` and `e` in a directed graph. The function operates over table `edges`, where a row `<h, t, w>`

[96]: Reynolds (1993), 'The discoveries of continuations'

[19]: Reynolds (1972), 'Definitional Interpreters for Higher-Order Programming Languages'

[97]: Danvy et al. (2001), 'Defunctionalization at work'

[98]: Gibbons (2021), 'Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity'

[99]: Schöpp (2013), 'On interaction, continuations and defunctionalization'

[100]: Danvy (2008), 'Defunctionalized interpreters for programming languages'

[101]: Gurevich (2000), 'Sequential abstract-state machines capture sequential algorithms'

[15]: Hirn et al. (2021), 'One WITH RECURSIVE is Worth Many GOTOs'

[27]: Ramachandra et al. (2018), 'Froid: Optimization of Imperative Programs in a Relational Database'

[102]: Lawson (2005), 'How Functions can Wreck Performance'

[103]: Floyd (1962), 'Algorithm 97: Shortest Path'

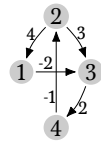
Figure 8.1.: Recursive UDF `floyd`, a SQL transcription of function `floyd` of Figure 8.2b. Yields NULL if there is no path from nodes `s` to `e` via nodes `1...n`.

```

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

```

Figure 8.2.: Floyd & Warshall’s algorithm over a directed graph (no negative cycles). We have $floyd(4, 2, 3) = 2$ and $floyd(3, 3, 2) = \infty$, for example.



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

$$floyd(0, s, e) = \begin{cases} w & \text{if } s \xrightarrow{w} e \\ \infty & \text{otherwise} \end{cases}$$

$$floyd(n, s, e) = \min(floyd(n-1, s, e), floyd(n-1, s, n) + floyd(n-1, n, e))$$

(a) Graph encoding.

(b) Algorithm in its recursive, textbook style.

represents the directed edge $h \xrightarrow{w} t$ of length w (Figure 8.2a shows a sample graph and its encoding in table `edges`).

Note that the code in Figure 8.1 is a direct translation of the recursive function `floyd` (see Figure 8.2b) into SQL. This *functional style* [104, 105] formulation results in a compact and readable SQL implementation of `floyd`, but causes a flood of recursive UDF calls during evaluation (the UDF of Figure 8.1 performs $\sum_{i=1}^n 3^i = \frac{(3^{n+1}-3)}{2}$ such calls in the absence of memoization). On each top-level or recursive call, the SQL engine creates a new plan context for callee `floyd` to

[104]: Duta et al. (2020), ‘Functional-Style SQL UDFs with a Capital ‘F’’

[105]: Duta (2022), ‘Another Way to Implement Complex Computations: Functional-Style SQL UDF’

[78]: Graefe (1994), ‘Volcano—An Extensible and Parallel Query Evaluation System’

- (P1) compile the `SELECT` block comprising the function’s body into a plan,
- (P2) improve this initial plan through optimizing plan rewrites,
- (P3) instantiate the resulting plan given the current arguments `n`, `s`, and `e`,
- (P4) evaluate the plan using a Volcano-style interpreter [78], and finally
- (P5) tear down plan data structures before the result is returned and the evaluation of the calling query’s plan can resume.

Since the engine needs to keep the plans for callers and callees around, the evaluation of any recursive UDF `f` leads to a nesting of plan contexts c_0, c_1, \dots as depicted in Figure 8.3. The repeated effort of plan generation and instantiation (steps P1 and P2, denoted *call* in Figure 8.3) plus teardown and resumption of the caller’s plan (P5, denoted *ret*) adds up to a significant runtime toll that can easily dwarf the productive time spent evaluating the plan for the body of `f` (steps P3 and P4, denoted *eval*). Plans are rich data structures and, in a sense, the engine finds itself creating and destroying “*super-heavy stack frames*” to drive the evaluation of recursive UDFs.

If we profile the query engine of PostgreSQL (version 15) during the evaluation of a call to `floyd`—which, in this particular case, results in 88,573 recursive calls—we find that the system spends 96% of the



Figure 8.3.: Nested plan contexts built to evaluate a top-level SQL query Q that contains a call to a linear-recursive UDF f . Overall evaluation time for Q is $t_\omega - t_\alpha$.

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

The situation is certainly dire, but PostgreSQL actually compares favorably to other off-the-shell SQL DBMSs: MySQL forbids the use of recursion in SQL UDFs (or *stored functions*) entirely [106, §25.8], while Oracle and Microsoft SQL Server impose restrictions such as recursion depth limits on UDFs (50 and 32, respectively). PostgreSQL will bail out when the stacked plan contexts exhaust the available process memory of the DBMS server [33, 35, 67]. The bottom line is that UDFs seem to be more of an afterthought in SQL engine design than anything else.

Goal: Treating SQL UDFs Like Functions (not Queries). Does the associated runtime penalty thus render the use of function-centric SQL code—and recursion, in particular—impractical? Since functional-style UDFs are an elegant way to express and perform *complex computation close to the data* [22, 104], we would consider this a real loss.

In this part of the thesis, we propose a compilation method that eliminates immediate (re-)planning on each recursive call. Instead, we treat recursive UDFs for what they are: *functions*. This opens up a box that contains tools other than the plan hammer:

- (F1) We consider a UDF f to be a plain function f in the sense of functional programming (FP). Function f operates on SQL data model values and embeds scalar SQL expressions, but is otherwise a vanilla function (Section 9.2).
- (F2) To f , we then apply a pipeline of established function compilation techniques, see Figure 8.4. Specifically, we translate f into CPS [57, 107], use defunctionalization [19], and finally transform f into trampolined style [10] (Sections 9.3 and 9.4).
- (F3) Function f in trampolined style implements a single loop which is readily expressed in terms of a recursive CTE, *i.e.*, an iterative query form that is widely supported by SQL DBMSs since the advent of the SQL:1999 standard [37, 38, 42]. We obtain SQL query Q_f , essentially an CTE-based interpreter loop for UDF f .

[106]: *MySQL 8.0 Documentation*

[33]: *PostgreSQL 15 Documentation*

[35]: *Microsoft SQL Server 2022 Documentation*

[67]: *Oracle Database PL/SQL Language Reference 21c*

[22]: Rowe et al. (1987), ‘The PostgreSQL Data Model’

[104]: Duta et al. (2020), ‘Functional-Style SQL UDFs with a Capital ‘F’’

[57]: Appel (1998), ‘SSA is Functional Programming’

[107]: Sussmann et al. (1975), ‘Scheme: An Interpreter for Extended Lambda Calculus’

[19]: Reynolds (1972), ‘Definitional Interpreters for Higher-Order Programming Languages’

[10]: Ganz et al. (1999), ‘Trampolined style’

[37]: Eisenberg et al. (1999), ‘SQL:1999, Formerly Known as SQL3’

[38]: *SQL:1999 Standard. Database Languages—SQL—Part 2: Foundation*

[42]: Finkelstein et al. (1996), *Expressive Recursive Queries in SQL*

- (F4) Q_f does not perform any recursive UDF calls and thus will be *planned once* in tandem with its enclosing SQL query. In addition, the compiled form provides hooks for a variety of optimizations—most notably memoization—that make the evaluation of Q_f significantly more efficient than the original UDF f , which Q_f can completely replace (Section 9.4).

The compilation stages in Figure 8.4 implement a source-to-source translation from recursive UDFs to CTEs that is non-invasive and applicable to any DBMS that adheres to SQL:1999—note that this even includes systems that do not natively support recursive UDFs (like MySQL). Chapter 10 applies this approach to UDF compilation to a set of recursive functions of varying complexity to demonstrate that function-centric SQL code indeed is one viable way to efficiently compute close to database-resident data.

The main focus of this part is on the marked area in Figure 8.4. Most of the compilation steps can be reused from Part 1 ‘Compiling PL/SQL Away’ with only minor local changes.

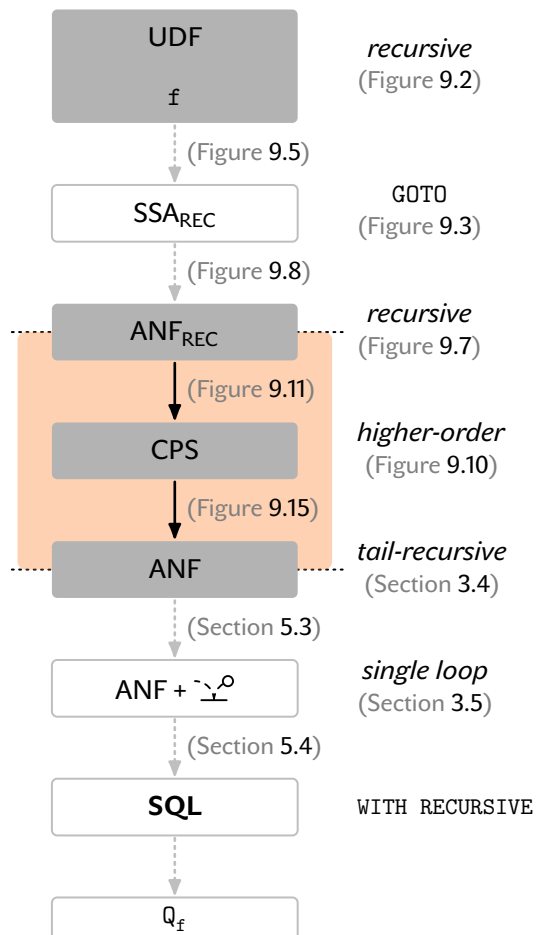


Figure 8.4.: Compilation stages and intermediate UDF forms.

Treating Recursive UDFs Like Functions 9.

In the following, we will unfold the details of SQL-to-SQL compilation of recursive UDFs into CTEs. We start with recursive SQL functions that follow the grammar definition in Figure 9.2. We assume that the UDF is already in this form and will not go into detail on how to do this. The reason for this is that we would need a full SQL grammar definition to properly define this transformation. This is not only beyond the scope of this thesis, but also does not add any new insight.

We will reuse some of the existing ideas from the PL/SQL compilation. Again, we use black boxing of SQL subexpressions to prepare the compilation of UDF f . This focuses the compilation on the essence of the recursive computation that f performs, *i. e.*, (1) the conditionals that separate base from recursive cases, and (2) the sites of recursive calls. These essentials are preserved while all other SQL expressions are wrapped in black boxes. The contents of these boxes have no effect on subsequent UDF compilation steps, and the SQL fragments they contain only reappear when the final CTE Q_f is generated. Consequently, the atomic types of this language are just the scalar SQL types.

Figure 9.1 shows the boxes Q_1, \dots, Q_3 and the scalar SQL expressions they contain for UDF f_{floyd} of Figure 8.1. Free variables and recursive call sites inside a box $Q[v_0, \dots, v_n]$ are exposed in terms of box parameters v_i : replacing v_i by e_i in Q yields the original SQL expression. (We abbreviate $Q_0[v_0]$ by v_0 and $Q[]$ by Q to aid readability.) Besides the boxes, we are left with the top-level `SELECT` block whose `CASE-WHEN-ELSE-END` conditional identifies the base and recursive cases in `floyd`.

```
1 -- length of shortest path (via nodes 1..n) from node s to e
2 CREATE FUNCTION floyd(n int, s int, e int) RETURNS int AS
3 $$
4   SELECT CASE WHEN (v0 = 0)[n]
5     THEN (SELECT edge.w
6           FROM edges AS edge
7           WHERE (edge.here, edge.there) = (v0, v1))[s, e]
8     ELSE LEAST(v0, v1 + v2)[floyd(n-1, s, e),
9              Q3[v1, v2]
10             floyd(n-1, s, n),
11             floyd(n-1, n, e)]
12   END;
13 $$ LANGUAGE SQL STABLE;
```

Figure 9.1.: UDF `floyd` and SQL subexpression boxes.

Figure 9.2.: Admissible SQL UDF dialect.

<code>udf ::= CREATE FUNCTION $f(\overline{v} \ \overline{\tau})$</code>	SQL UDF
<code> RETURNS [SETOF] τ AS</code>	
<code> q LANGUAGE SQL STABLE;</code>	
<code> q ::= SELECT e;</code>	UDF body
<code> e ::= CASE \overline{w} ELSE e END</code>	conditional
p	
p ::= a	
$f(\overline{p})$	recursive UDF call
<code> w ::= WHEN e THEN e</code>	
<code> a ::= SQL query [\overline{p}]</code>	boxed SQL query
<code> v ::= <identifier></code>	variable/function name
<code> f ::= <name of recursive SQL UDF></code>	
<code> τ ::= <scalar SQL type></code>	scalar value type

```

f ::= fun v( $\overline{v} \ \overline{\tau}$ ) : [SETOF]  $\tau$ 
    { $b$ }
b ::=  $\kappa$  :  $p$  s;
p ::= v :  $\tau$   $\leftarrow$   $\phi(\kappa:v, \dots, \kappa:v)$ ;
    p
    |  $\varepsilon$ 
s ::= v  $\leftarrow$  a
    | REC v = v( $\overline{a}$ )
    | IF v THEN t ELSE t
    | t
    | EMIT a
    | s; s
t ::= GOTO  $\kappa$ 
    | RETURN a
a ::= SQL query [ $\overline{v}$ ]
v ::= <identifier>
 $\tau$  ::= <scalar SQL type>
 $\kappa$  ::= <block label>

```

Figure 9.3.: GOTO-based imperative intermediate form extended with explicit binding for recursive calls.

[53]: Rastello et al. (2022), *SSA-based Compiler Design*

```

-- Input expression:
f(f(f(Q[p1, ..., pn])))

-- Resulting program in SSA:
v1  $\leftarrow$  p1;
:
vn  $\leftarrow$  pn;
vq  $\leftarrow$  Q[v1, ..., vn]
REC vr1 = f(vq);
REC vr2 = f(vr1);
REC vr3 = f(vr2);
RETURN vr3;

```

Figure 9.4.: Example of applying Rule PROXY and auxiliary rules $\Rightarrow_{\mathcal{R}}$ to a nested expression.

9.1. Translation from SQL to SSA_{REC}

The first compilation step is to convert the recursive SQL UDF to SSA_{REC} form as shown in Figure 9.3. The specifics of ϕ -function placement will not be discussed here, as this is a very common technique. We refer the reader to the literature on SSA for more details [53].

Again, we want to emphasize that this and all following compilation steps leave the boxes \mathbb{Q} intact: in particular, we are never concerned with the SSA-equivalent of SQL's SELECT-FROM-WHERE blocks (as contained in box \mathbb{Q}_2 , for example). The boxes are not unpacked before we reach the end of the translation pipeline and are ready to assemble the recursive CTE.

The inference rules for compiling from recursive SQL UDFs to SSA consists of $\mapsto_{\mathcal{R}}$, and auxiliaries $\Rightarrow_{\mathcal{R}}$ and $\Leftarrow_{\mathcal{R}}$. Compilation starts with the top-level Rule UDF, which takes a recursive SQL UDF and returns a GOTO-based program. The body of the UDF is mapped into a dictionary s of blocks, which contain statements of the simple imperative GOTO-based form defined in Figure 9.3. Each block in dictionary s is identified by its label κ . This *label-to-block* dictionary can be updated via $s_2 \equiv s_1 +_{\kappa} [\langle \text{statements} \rangle]$. The block κ_1 is created if it does not already exist in s_1 . Otherwise, the statements are appended to the existing block κ_1 . The relation $\Gamma \vdash \langle c \mid \kappa_1 \mid s_1 \rangle \mapsto_{\mathcal{R}} \langle \kappa_2 \mid s_2 \rangle$ is the core of the rules $\mapsto_{\mathcal{R}}$ and $\Rightarrow_{\mathcal{R}}$. It transforms the input c into a sequence of simple imperative statements. After c has been compiled, compilation continues using the block label κ_2 . This is identical to the inference rule setup described in Section 5.1.

The recursive SQL UDF dialect allows arbitrary nesting of black-box parameters and recursive call parameters. This is encoded in the grammar's non-terminals p and a in Figure 9.2. To establish the SSA property and for later compilation steps, this nesting must be eliminated and flattened with variable assignments $v \leftarrow a$ and recursive binds `REC v = v(\overline{a})`. This flattening transformation sequentializes computations and thereby ensures that the parameters of the black boxes will always be values. The auxiliary $\Rightarrow_{\mathcal{R}}$ performs this transformation. This auxiliary defines a slightly different relation $\Gamma \vdash \langle c \mid \kappa_1 \mid s_1 \rangle \Rightarrow_{\mathcal{R}} \langle \kappa_2 \mid s_2 \mid v \rangle$, which returns the binding of the final

$$\begin{array}{c}
\frac{\emptyset \vdash \langle e \mid \text{start} \mid [] \rangle \mapsto_{\mathcal{R}} \langle \kappa_1 \mid s_1 \rangle \quad \text{blocks} \equiv \bigcup_{\kappa \in s_1} \kappa : \varepsilon \ s_1[\kappa]}{\text{(UDF)}} \\
\frac{\begin{array}{l} \text{CREATE FUNCTION } f(\overline{v} \ \tau) \\ \text{RETURNS } [\text{SETOF}] \ \tau_r \ \text{AS} \\ \text{\$ \$ SELECT } e \ \text{\$ \$} \\ \text{LANGUAGE SQL STABLE;} \end{array} \quad \mapsto_{\mathcal{R}} \quad \begin{array}{l} \text{fun } f(\overline{v} \ \tau) : [\text{SETOF}] \ \tau_r \{ \\ \text{blocks} \\ \} \end{array}}{\text{(CASE)}} \\
\frac{\Gamma \vdash \langle \overline{w} \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa_{\text{when}} \mid s_1 \rangle \quad \Gamma \vdash \langle e \mid \kappa_{\text{when}} \mid s_1 \rangle \mapsto_{\mathcal{R}} \langle \kappa_{\text{else}} \mid s_2 \rangle}{\Gamma \vdash \langle \text{CASE } \overline{w} \ \text{ELSE } e \ \text{END} \mid \kappa \mid s \rangle \mapsto_{\mathcal{R}} \langle \kappa_{\text{else}} \mid s_2 \rangle} \\
\frac{\begin{array}{l} \kappa_{\text{then}}, \kappa_{\text{else}}, \kappa_{\text{meet}} \equiv \text{new block labels} \quad p \equiv \text{new var} \\ \Gamma \vdash \langle e_1 \mid \kappa \mid s \rangle \mapsto_{\mathcal{R}} \langle \kappa_1 \mid s_1 \rangle \\ \Gamma \vdash \langle e_2 \mid \kappa_1 \mid s_1 \rangle \mapsto_{\mathcal{R}} \langle \kappa_2 \mid s_2 \rangle \\ \Gamma \vdash \langle \overline{w} \mid \kappa_2 \mid s_2 \rangle \Rightarrow_{\mathcal{R}} \langle \kappa_3 \mid s_3 \rangle \\ b \equiv [p \leftarrow q; \text{IF } p \ \text{THEN GOTO } \kappa_{\text{then}} \ \text{ELSE GOTO } \kappa_{\text{else}};] \\ s_3 \equiv s_2 +_{\kappa} b +_{\kappa_1} [\text{GOTO } \kappa_{\text{meet}};] +_{\kappa_2} [\text{GOTO } \kappa_{\text{meet}};] \end{array}}{\Gamma \vdash \langle \text{WHEN } e_1 \ \text{THEN } e_2 \ \overline{w} \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa_3 \mid s_3 \rangle} \text{(WHEN1)} \\
\frac{}{\Gamma \vdash \langle \varepsilon \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa \mid s \rangle} \text{(WHEN2)} \\
\frac{\Gamma \vdash \langle p \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa \mid s_1 \mid p_1 \rangle \quad s_2 \equiv s_1 +_{\kappa} [\text{RETURN } p_1]}{\Gamma \vdash \langle p \mid \kappa \mid s \rangle \mapsto_{\mathcal{R}} \langle \kappa \mid s_2 \rangle} \text{(PROXY)} \\
\frac{\Gamma \vdash \langle \overline{p} \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa \mid \overline{s} \mid \overline{p}_1 \rangle \quad v \equiv \text{new var} \quad s_1 \equiv \overline{s} +_{\kappa} [v \leftarrow \mathbf{Q}[\overline{p}_1];]}{\Gamma \vdash \langle \mathbf{Q}[\overline{p}] \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa \mid s_1 \mid v \rangle} \text{(BOX)} \\
\frac{\begin{array}{l} \Gamma \vdash \langle \overline{p} \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa \mid \overline{s} \mid \overline{p}_1 \rangle \\ v \equiv \text{new var} \quad s_1 \equiv \overline{s} +_{\kappa} [\text{REC } v = f(\overline{p}_1);] \end{array}}{\Gamma \vdash \langle f(\overline{p}) \mid \kappa \mid s \rangle \Rightarrow_{\mathcal{R}} \langle \kappa \mid s_1 \mid v \rangle} \text{(CALL)}
\end{array}$$

Figure 9.5.: Translation from recursive SQL UDFs to SSA.

unnested expression. Rule PROXY plays a special role in this part of the transformation. The rule is the entry point for such nested expressions and produces a block terminal RETURN v , where v is the final generated bound identifier. Figure 9.4 shows an example of these rules in action.

If we apply this set of inference rules to UDF `floyd` in Figure 9.1, we get the SSA variant in Figure 9.6. Note that we have made some simplifications that are common practice. This is done to improve readability. Also, note that we have applied the ϕ -function placement transformation to the SSA program. The SSA program is still recursive, but the recursive calls are now explicitly marked with the REC keyword. Also, the recursive calls are now sequentialized and no longer nested. This is important for the next step in the compilation pipeline.

```

1 fun floyd(n int, s int, e int) : int {
2   start:
3   v0 ← Q1[n];
4   IF v0
5   THEN RETURN Q2[s,e];
6   ELSE GOTO κthen;
7   κthen:
8   n ← φ(start:n);
9   s ← φ(start:s);
10  e ← φ(start:e);
11  REC v1 = floyd(n-1, s, e);
12  REC v2 = floyd(n-1, s, n);
13  REC v3 = floyd(n-1, n, e);
14  RETURN Q3[v1,v2,v3];
15 }

```

Figure 9.6.: Function `floyd` after translation to SSA_{REC} .

9.2. Transition to ANF_{REC}

The input UDF f is now transformed from SSA_{REC} to a first-order function \bar{f} expressed in a simple ANF_{REC} -style language (see Figure 9.7). This language is closely related to the ANF language used in the first part of this thesis (recall Section 3.4). But it is extended with explicit recursive function calls: $\text{REC } v = v(\bar{a}) \text{ IN } e$. Importantly, since SQL subexpression boxing has left us with the recursive backbone of the UDF, (1) **IF-THEN-ELSE** conditionals, (2) function invocations, and (3) the boxed expressions themselves already make a complete target language. However, we do have support for a few more syntax constructs to handle recursive PL/SQL functions as well.

The translation from SSA_{REC} to ANF_{REC} is exactly the same as described in Sections 3.4 and 5.2. This set of inference rules can be reused as is. However, since we have extended the SSA grammar to include recursive binds, we need to add the additional inference rule in Figure 9.8 to handle this case. For UDF `floyd`, the resulting function is reproduced in Figure 9.9. We have again applied some common program simplifications. Most notably, function inlining.

```

p ::= fun v(τ̄) : [SETOF] τ {f̄}
f ::= v(τ̄) : τ = e
e ::= a
   | v(ā)
   | IF v THEN e ELSE e
   | LET v = a IN e
   | EMIT a; e
   | REC v = v(ā) IN e
a ::= SQL query [τ̄]
v ::= <identifier>
τ ::= <scalar SQL type>

```

Figure 9.7.: Intermediate functional language in ANF extended with explicit binding for recursive calls.

Figure 9.8.: Additional translation rule for the Chakravarty, Keller, and Zadarnowski-style SSA to ANF translation in Section 5.2.

$$\frac{\Gamma \vdash s \mapsto_{\mathcal{A}} s_1}{\Gamma \vdash \text{REC } v = a; s \mapsto_{\mathcal{A}} \text{REC } v = a \text{ IN } s_1} (\text{REC})$$

```

1 fun floyd(n int, s int, e int) : int {
2   LET v0 =  $\kappa_1$ [n] IN
3   IF v0
4   THEN
5      $\kappa_2$ [s,e]
6   ELSE
7     REC v1 = floyd(n-1, s, e) IN
8     REC v2 = floyd(n-1, s, n) IN
9     REC v3 = floyd(n-1, n, e) IN
10     $\kappa_3$ [v1,v2,v3];
11 }

```

Figure 9.9.: Function `floyd` after translation from SSA_{REC} to ANF_{REC} . Function inlining has been applied.

9.3. From Recursion Towards Iteration: CPS and Defunctionalization

Now that we have a direct-style ANF representation of the program with explicit recursive calls, we need to remove all recursive function calls and replace them with tail calls. This problem is very well studied by the programming language community and can be archived using the CPS transformation [108–111]. Programs in CPS are *higher-order* and perform only tail calls. Later, we will translate these tail calls into iteration. CPS also explicitly orders the evaluation of function arguments. We will need this property later when we instantiate the CTE-based interpreter. The function `floyd` in CPS (see Figure 9.12) computes the intermediate results `v1`, `v2`, and `v3` (in that order) and passes them to the continuations κ_1 , κ_2 , and κ_3 , respectively.

Although CPS is very well researched, many publications only show translation rules for the simple *lambda calculus* (as is common practice in the programming language community), if there are any rules at all. So we have to define the transformation ourselves if we want to apply it to our own ANF IR. The CPS grammar shown in Figure 9.10, and the CPS translation rules are loosely based on [112]. The explicit encoding of continuations in the grammar is the most important aspect of their CPS grammar. This makes the CPS translation (see Figure 9.11), and the following defunctionalization step easy to define and reason about.

f	::=	def $v(\bar{v} \bar{\tau} \mid \kappa : \neg\tau) : \tau \{ \bar{t} \}$	top-level function
t	::=	$\kappa(a)$	jump
		IF v THEN t ELSE t	
		LET $v = a$ IN t	
		cnt $\kappa(v \tau) \{ t \}; v(\bar{a} \mid \kappa)$	continuation
		EMIT $a; t$	emit value
a	::=	SQL query $[\bar{v}]$	boxed SQL query
v	::=	$\langle \text{identifier} \rangle$	variable/function name
κ	::=	$\langle \text{identifier} \rangle$	continuation identifier
τ	::=	$\langle \text{scalar SQL type} \rangle$	parameter/return type
		$\neg\tau$	continuation type

[108]: Appel (2007), *Compiling with continuations*

[109]: Kennedy (2007), ‘Compiling with continuations, continued’

[110]: Cong et al. (2019), ‘Compiling with Continuations, or without? Whatever.’

[111]: Paraskevopoulou et al. (2021), ‘Compiling with continuations, correctly’

[112]: Müller et al. (2023), ‘Back to Direct Style: Typed and Tight’

Figure 9.10.: Intermediate functional language in CPS.

Figure 9.11.: Translation to continuation passing style.

$$\begin{aligned}
\mathcal{C}[\text{IF } v \text{ THEN } e_0 \text{ ELSE } e_1]_{\kappa} &= \text{IF } v \text{ THEN } \mathcal{C}[e_0]_{\kappa} \text{ ELSE } \mathcal{C}[e_1]_{\kappa} \\
\mathcal{C}[\text{REC } v = a \text{ IN } e]_{\kappa} &= \text{cnt } \kappa_0(v) \{ \mathcal{C}[e]_{\kappa} \}; \mathcal{C}[a]_{\kappa_0} \quad \text{where } \kappa_0 \text{ fresh} \\
\mathcal{C}[v(a)]_{\kappa} &= v(a \mid \kappa) \\
\mathcal{C}[\text{fun } v(\bar{v}) \{e_0\}; e]_{\kappa} &= \text{def } v(\bar{v} \mid \kappa_0) \{ \mathcal{C}[e_0]_{\kappa_0} \}; \mathcal{C}[e]_{\kappa} \quad \text{where } \kappa_0 \text{ fresh} \\
\mathcal{C}[\text{EMIT } a; t]_{\kappa} &= \text{EMIT } a; \mathcal{C}[t]_{\kappa} \\
\mathcal{C}[a]_{\kappa} &= \kappa(a) \\
\mathcal{C}[\text{LET } v = a \text{ IN } e]_{\kappa} &= \text{LET } v = a \text{ IN } \mathcal{C}[e]_{\kappa}
\end{aligned}$$

```

1 def floyd(n int, s int, e int | κ₀ : stack) : int {
2   LET v0 = Q₁[n] IN
3   IF v0
4   THEN
5     κ₀(Q₂[s,e])
6   ELSE
7     cnt κ₁(v1) {
8       cnt κ₂(v2) {
9         cnt κ₃(v3) {
10          κ₀(Q₃[v1,v2,v3])
11          }; floyd(n-1, n, e | κ₃)
12          }; floyd(n-1, s, n | κ₂)
13          }; floyd(n-1, s, e | κ₁)
14   }

```

Figure 9.12.: Function `floyd` after translation from ANF_{REC} to CPS.

Continuations As Data: Defunctionalization. CPS leaves us with a higher-order program representation. But functions are not first-class in the SQL domain, which means we have to eliminate all higher-order functions to get back to SQL. Although CPS is a whole program transformation, it is important to note that only the continuations created by the CPS transformation are truly new and higher-order. So we can selectively eliminate those. We choose to represent the higher-order continuations in terms of data. Figure 9.16 shows the function `floyd` after this transformation.

[19]: Reynolds (1972), ‘Definitional Interpreters for Higher-Order Programming Languages’

Defunctionalization [19] introduces closure records $\langle 1, env \rangle$ in which tag 1 identifies the continuation (for `floyd`, $k \in \{\kappa_1, \kappa_2, \kappa_3\}$) and env holds the environment of free variables. For `floyd`, the environment is $env \equiv n, s, e, v1, v2, v3$. We replace a variable v with `NULL` if v is undefined in its environment, and thus get fixed-width closure records (1 + 5 = 6 in the case of `floyd`). This replacement is done in Rule `DEFUN CONT` of Figure 9.15. The nesting of continuations is encoded in terms of a stack of closure records (see argument κs with operations `EMPTY`, `PUSH`, `POP`, `TOP` in Figure 9.16). The auxiliary function `apply(..., x, κ)` examines the tag 1 of the topmost closure record on the stack κ and invokes the corresponding continuation on the argument x . If `apply` detects that the continuation stack κ is empty, the final result x is returned (see Line 14 of Figure 9.16). So we can start the computation with an empty stack.

Figure 9.15 shows this translation from CPS back to a direct-style representation in ANF. The required ANF grammar is shown in Figure 9.14 and is equivalent to the ANF grammar used in Part 1 of this thesis (remember Section 3.4), except for the dispatching `CASE` extension. The inference rules consist of \mapsto_{df} and auxiliary \Rightarrow_{df} which define the relation $params \mid \Gamma \vdash t \mapsto_{df} \langle e \mid a \rangle$. A CPS term t is transformed into an ANF expression e . If t is a continuation, an additional `apply` branch a is returned (see Rule `DEFUN CONT`). The context of the inference rules $params \mid \Gamma$ is divided into the fixed

set of function parameters *params* and the set of *bound* variables Γ . These sets are initially equal. The translation of a CPS program starts with the Rule DEFUN DEF. This rule instantiates the `apply` and `start` function templates and creates the initial translation context (see $\bar{v} \mid \bar{v} \vdash t$). Most of the heavy lifting is done in Rule DEFUN CONT. This rule translates a higher-order continuation back to a first-order function call with an explicit stack. The stack contains closure records in the form of tuples, based on the set of *used* variables (for the `floyd` function, this set contains the variables $\{n, s, e, v1, v2, v3\}$). If a variable is not (yet) bound, *i. e.*, not contained in the set Γ , it is replaced by `NULL`.

The naming of the continuation stack variable is a subtle but important detail of this rule set. For the original function, the stack is named κ_s , while for `apply` it is named κ . This naming scheme simplifies the stack management in the defunctionalized version of the program (for function `floyd`, see Lines 8, 9 and 19 to 21). The variable κ_s is defined as the `TAIL` of the stack in the `apply` function template. This ensures that the top element of the stack is either replaced or removed during execution. The original function keeps the stack as it is, perhaps pushing an additional element. Figure 9.13 shows the call graph of function `floyd` after defunctionalization. Regular arrows \rightarrow indicate that the stack will remain unchanged. A stack `PUSH` operation is marked with \rightarrow , and the removal or replacement of the top element is marked with arrows \rightarrow .

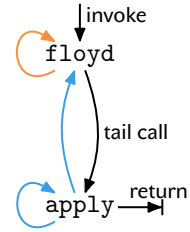


Figure 9.13.: Call graph after defunctionalization.

```

p ::= fun v( $\bar{v}$   $\tau$ ) : [SETOF]  $\tau$  { $\bar{f}$ }
f ::= v( $\bar{v}$   $\tau$ ) :  $\tau$  = e
e ::= a
    | v( $\bar{a}$ )
    | IF v THEN e ELSE e
    | LET v = a IN e
    | EMIT a; e
    | CASE v OF 'v' : e
a ::= SQL query [ $\bar{v}$ ]
v ::= <identifier>
 $\tau$  ::= <scalar SQL type>
    
```

Figure 9.14.: Intermediate functional language in ANF.

$$\begin{array}{c}
 \bar{v} \mid \bar{v} \vdash t \Rightarrow_{\mathcal{DF}} \langle t_1 \mid \text{branches} \rangle \quad \text{vars} \equiv \text{fv}(\text{branches}) \quad \text{apply} \equiv \\
 \begin{array}{l}
 \text{fun start}(\bar{v}) \{ \\
 \quad f(\bar{v}, \text{NULL}, \text{EMPTY_STACK}()) \\
 \} \\
 \\
 \text{fun apply}(\bar{v}, x, k) \{ \\
 \quad \text{IF EMPTY}(k) \\
 \quad \text{THEN } x \\
 \quad \text{ELSE LET } \langle l, \text{vars} \rangle = \text{TOP}(k) \text{ IN} \\
 \quad \quad \text{LET } \text{ks} = \text{TAIL}(k) \text{ IN} \\
 \quad \quad \text{CASE 1 OF} \\
 \quad \quad \quad \text{branches} \\
 \}
 \end{array} \\
 \hline
 \vdash \text{def } f(\bar{v} \mid \kappa_i) \{ t \} \mapsto_{\mathcal{DF}} \left\langle \text{fun } f(\bar{v}, x, \text{ks}) \{ t_1 \} \mid \text{apply} \right\rangle \quad (\text{DEFUN DEF}) \\
 \\
 \text{values} \equiv \left[\begin{array}{ll} x & \text{if } v_k = u \\ u & \text{if } u \in \Gamma \\ \text{NULL} \square & \text{otherwise} \end{array} \middle| u \in \text{used}(t) \right] \quad \text{tuple} \equiv (' \kappa_i ', \text{values}) \\
 \text{vs} \mid v_k, \Gamma \vdash t \Rightarrow_{\mathcal{DF}} \langle f \mid \text{branches} \rangle \quad \text{branch} \equiv ' \kappa_i ': f \\
 \hline
 \text{vs} \mid \Gamma \vdash \text{cnt } \kappa_i(v_k) \{ t \}; \Rightarrow_{\mathcal{DF}} \langle v(\bar{a}, \text{NULL} \square, \text{PUSH}([\text{tuple}], \text{ks})) \mid \text{branch } \text{branches} \rangle \quad (\text{DEFUN CONT}) \\
 v(\bar{a} \mid \kappa_i) \\
 \\
 \hline
 \text{vs} \mid v, \Gamma \vdash \kappa_i(e) \Rightarrow_{\mathcal{DF}} \langle \text{apply}(\text{vs}, e[v \mapsto x], \text{ks}) \mid \rangle \quad (\text{DEFUN JUMP}) \\
 \\
 \hline
 \text{vs} \mid \Gamma \vdash t_0 \Rightarrow_{\mathcal{DF}} \langle p_0 \mid \text{branches}_0 \rangle \quad \text{vs} \mid \Gamma \vdash t_1 \Rightarrow_{\mathcal{DF}} \langle p_1 \mid \text{branches}_1 \rangle \\
 \text{vs} \mid \Gamma \vdash \text{IF } v \text{ THEN } t_0 \text{ ELSE } t_1 \Rightarrow_{\mathcal{DF}} \langle \text{IF } v \text{ THEN } p_0 \text{ ELSE } p_1 \mid \text{branches}_0 \text{ branches}_1 \rangle \quad (\text{DEFUN IF}) \\
 \\
 \hline
 \text{vs} \mid \Gamma \vdash t \Rightarrow_{\mathcal{DF}} \langle p \mid \text{branches} \rangle \\
 \text{vs} \mid \Gamma \vdash \text{EMIT } a; t \Rightarrow_{\mathcal{DF}} \langle \text{EMIT } a; p \mid \text{branches} \rangle \quad (\text{DEFUN EMIT}) \\
 \\
 \hline
 \text{vs} \mid v, \Gamma \vdash e \Rightarrow_{\mathcal{DF}} \langle p \mid \text{branches} \rangle \\
 \text{vs} \mid \Gamma \vdash \text{LET } v = a \text{ IN } e \Rightarrow_{\mathcal{DF}} \langle \text{LET } v = a \text{ IN } p \mid \text{branches} \rangle \quad (\text{DEFUN LET})
 \end{array}$$

Figure 9.15.: Translation from CPS back to direct-style ANF.

```

1 fun start(n int, s int, e int) : int {
2   floyd(n, s, e, NULL, EMPTY_STACK())
3 }
4
5 fun floyd(n int, s int, e int, x int, ks stack) : int {
6   LET v0 = Q1[n] IN
7   IF v0
8   THEN apply(n, s, e, Q2[s,e], ks)
9   ELSE floyd(n-1, s, e, NULL, PUSH(['k1',n,s,e,NULL,NULL], ks))
10 }
11
12 fun apply(n int, s int, e int, x int, ks stack) : int {
13   IF EMPTY(ks)
14   THEN x
15   ELSE
16     LET <l, n, s, e, v1, v2, v3> = TOP(ks) IN
17     LET ks = TAIL(ks) IN
18     CASE 1 OF
19       'k1': floyd(n-1, s, n, NULL, PUSH(['k2',n,s,e,x,NULL], ks))
20       'k2': floyd(n-1, n, e, NULL, PUSH(['k3',n,s,e,v1,x], ks))
21       'k3': apply(n, n, e, Q3[v1,v2,x], ks)
22 }
    
```

Figure 9.16.: Function floyd after translation from CPS back to direct-style ANF.

9.4. Trampoline Style: A Single Loop Replaces Mutual Recursion

After applying CPS and defunctionalization, we have arrived at the mutually *tail-recursive* pair of functions `f/apply`. This form of the program now allows us to reuse the entire tail of the existing compilation pipeline from the first part of this thesis (Sections 3.5, 3.6, 5.3 and 5.4). Therefore, the same arguments, considerations, and inference rules apply. This is an important property of our approach and shows that the trampolined style translation is a powerful and general technique for compiling both iterative and recursive programs to SQL. The only difference is that we now have to maintain the continuation stack explicitly.

From ANF to Trampoline Style ANF. The complexity of the call graph in Figure 9.13 is at odds with the single-loop iteration that SQL’s recursive CTEs can express (recall Section 3.1). A better fit is trampolined style, where a designated `trampoline` function is responsible for handling *all* function calls in a given program: to call `g` from `f`, (1) `f` invokes `trampoline`, with the arguments to be passed to `g` along with a function label `l = 'g'`, (2) then `trampoline` calls `g` as directed by `l`. The `trampoline`’s full control over whether and how calculations are performed allows for a wide variety of trampolined-style applications. Here, we are primarily interested in the inherent call graph simplification it provides (see Figure 9.17).

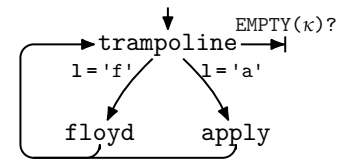


Figure 9.17.: Trampoline style (before inlining).

$$\frac{e \mapsto_f t}{\text{CASE } v \text{ OF } 'v' : e \mapsto_f \text{ CASE } v \text{ OF } 'v' : t} \text{ (CASE)}$$

Figure 9.18.: Additional translation rule for the ANF to trampolined style translation in Section 5.3.

Applying the inference rules of Section 5.3, extended by Rule CASE in Figure 9.18, to the function `floyd` after defunctionalization (Figure 9.16) results in the trampolined version of `floyd` shown in Figure 9.19. Note that the function parameter κ of the `apply` function has been replaced by κs using capture-free substitution. This is necessary because the trampolined-style transformation requires the same function parameters for the `f/apply` function family.

From Trampoline Style ANF to SQL. The final compilation step is to instantiate the `WITH RECURSIVE`-based trampolined-style SQL interpreter. Figure 9.20 shows the CTE we get from a straightforward translation of the `f/apply` function pair into SQL using the inference rules in Section 5.4. Note that we have kept the black boxes in place and the stack handling abstract for now. Here, the CTE is wrapped in a SQL UDF `floyd` that could replace the original in Figure 8.1. However, the CTE body in Lines 3 to 36 can also stand alone: it contains *no recursive calls* and could therefore be inlined at the call sites of `floyd`.

Just like the `trampoline` function, the recursive CTE works on tuples `(rec?, call, res, n, s, e, x, κs)` (in form of the working table). To start the interpretation, `Qinit` (*i.e.*, the `SELECT` of Line 4) places an

```

1 fun start(n int, s int, e int) : int {
2   trampoline(true, 'floyd', NULL, n, s, e, NULL, EMPTY_STACK())
3 }
4
5 fun trampoline(rec? bool, call text, res int, n int, s int, e int, x int, κs stack) : int {
6   IF NOT rec? THEN
7     res
8   ELSE
9     CASE call OF
10    'floyd':
11      LET v0 = Q1[n] IN
12      IF v0
13      THEN trampoline(true, 'apply', NULL, n, s, e, Q2[s,e], κs)
14      ELSE trampoline(true, 'floyd', NULL, n-1, s, e, NULL, PUSH(['κ1',n,s,e,NULL,NULL], κs))
15    'apply':
16      IF EMPTY(κs)
17      THEN trampoline(false, NULL, x, NULL, NULL, NULL, NULL, NULL)
18      ELSE
19        LET ⟨l, n, s, e, v1, v2, v3⟩ = TOP(κs) IN
20        LET κs1 = TAIL(κs) IN
21        CASE l OF
22          'κ1': trampoline(true, 'floyd', NULL, n-1, s, n, NULL, PUSH(['κ2',n,s,e,x,NULL], κs1))
23          'κ2': trampoline(true, 'floyd', NULL, n-1, n, e, NULL, PUSH(['κ3',n,s,e,v1,x], κs1))
24          'κ3': trampoline(true, 'apply', NULL, n, n, e, Q3[v1,v2,x], κs1)
25 }

```

Figure 9.19.: Function `floyd` after translation to trampolined style.

appropriate tuple—or: “instruction”—in the working table. The iterated query q_m in Lines 6 to 35 reads the current instruction tuple r from table `run`, processes it, and outputs the next instruction that (1) replaces the current tuple in `run` and (2) is also added to the union table `u`. Processing these instructions entails:

- (IP1) accessing the topmost continuation on stack κ_s . Similar to the LET in Line 19 of Figure 9.19, we use a `LATERAL` join to bind the tuple entries of the continuation to names and make them available to the rest of the query. There are a number of possible implementation alternatives on the SQL side for the stack κ_s and its `PUSH`, `TAIL`, `TOP`, `EMPTY` operations. We will return to this later in Section 9.6.
- (IP2) Then, by checking the function call $r.call \in \{\text{apply}, \text{floyd}\}$ and the closure tag $l \in \{\kappa_1, \kappa_2, \kappa_3\}$, the correct next instruction is selected.

Function `trampoline` implements step (IP2) in the form of `CASE-OF` multi-way conditionals. Here we use predicated `SELECT-WHERE` SQL query blocks concatenated with `UNION ALL`. Note that the `WHERE` predicates are mutually exclusive, so that at most one block can output an instruction tuple per iteration. For contemporary RDBMSs, this results in efficient physical execution plans (recall Section 5.4).

The assembly of the instruction tuples themselves directly mimics the `trampoline` function (e.g., Line 24 of Figure 9.20 corresponds to Line 22 of Figure 9.19). Once we unfold the contained black boxes, we obtain a syntactically complete CTE that can replace the original UDF of Figure 8.1. This concludes the compilation of recursive UDFs to SQL.

```

1 CREATE FUNCTION floyd(n int, s, int, e int) RETURNS int AS
2 $$
3 WITH RECURSIVE run("rec?", call, res, n, s, e, x, κs) AS (
4     SELECT true, 'floyd', NULL, n, s, e, NULL, EMPTY_STACK()
5     UNION ALL -- recursive UNION
6     SELECT "case1".*
7     FROM run AS r,
8     LATERAL (SELECT "cond1".*
9             FROM (Q1[n]) AS "if1"("v0"),
10            LATERAL (SELECT true, 'apply', NULL, n, s, e, Q2[s,e], κs
11                   WHERE "if1"."v0"
12                   UNION ALL
13                   SELECT true, 'floyd', NULL, n-1, s, e, NULL, PUSH(['κ1',n,s,e,NULL,NULL]), κs)
14                   WHERE NOT "if1"."v0") AS "cond1"
15     WHERE r.call = 'floyd'
16     UNION ALL
17     SELECT "cond2".*
18     FROM (SELECT false, NULL, x, NULL, NULL, NULL, NULL, NULL
19          WHERE EMPTY(κs)
20          UNION ALL
21          SELECT "case2".*
22          FROM (SELECT TOP(κs).*) AS _1(1, n, s, e, v1, v2, v3),
23          LATERAL (SELECT TAIL(κs)) AS _2(κs1),
24          LATERAL (SELECT true, 'floyd', NULL, n-1, s, n, NULL, PUSH(['κ2',n,s,e,x,NULL]), κs1)
25                  WHERE l = 'κ1'
26                  UNION ALL
27                  SELECT true, 'floyd', NULL, n-1, n, e, NULL, PUSH(['κ3',n,s,e,v1,x]), κs1)
28                  WHERE l = 'κ2'
29                  UNION ALL
30                  SELECT true, 'apply', NULL, n, n, e, Q3[v1,v2,x], κs1)
31                  WHERE l = 'κ3'
32          ) AS "case2"
33     WHERE NOT EMPTY(κs) AS "cond2"
34     WHERE r.call = 'apply') AS "case1"
35     WHERE r."rec?"
36 ) SELECT run.res FROM run WHERE NOT run."rec?"
37 $$ LANGUAGE SQL STABLE;

```

Figure 9.20.: Iterative CTE-based interpreter replacing the UDF `floyd` of Figure 8.1.

Union Table \equiv Instruction Trace. Given the semantics of recursive CTEs, each invocation of this SQL-based interpreter returns a union table `u` that collects a trace of all instructions evaluated by the interpreter. Each iteration contributes one row to `u`. To illustrate, Figure 9.21 shows a portion of the table resulting from a `floyd(2,2,3)` call (ignore the annotations \bullet for now). As expected, we find rows with `l = floyd`, representing the recursive calls to `floyd` (see Figure 9.19). Rows with `l = apply` correspond to the application of the current top continuation on stack `κs` to the intermediate result `x` (again, recall the invocations of `apply` in Figure 9.19). The last row with `call = NULL` and `rec? = false` holds the overall result value in column `res`. Exactly this (gray) table cell is extracted and returned by the final `SELECT` block in Line 36 of Figure 9.20.

run								
	rec?	call	res	n	s	e	x	KS
	true	floyd	NULL	2	2	3	NULL	
	true	floyd	NULL	1	2	3	NULL	
	true	floyd	NULL	0	2	3	NULL	
②	true	apply	NULL	NULL	NULL	NULL	3	[k ₁ , .]
③	true	apply	NULL	NULL	NULL	NULL	4	
④	true	apply	NULL	NULL	NULL	NULL	-2	
①	true	apply	NULL	NULL	NULL	NULL	2	
	true	floyd	NULL	1	2	2	NULL	
⑥	true	apply	NULL	NULL	NULL	NULL	NULL	

	true	floyd	NULL	0	1	3	NULL	
⑫	true	apply	NULL	NULL	NULL	NULL	-2	
⑩	true	apply	NULL	NULL	NULL	NULL	2	[k ₂ , .]
⑨	true	apply	NULL	NULL	NULL	NULL	2	
	false	NULL	2	NULL	NULL	NULL	NULL	

Figure 9.21.: CTE union table result for floyd(2,2,3).

9.5. Memoizing the Results of Recursive Calls

A reduction of function call overhead is welcome, and Chapter 10 will evaluate the performance advantage of the iterative interpreter over recursive UDF evaluation. However, avoiding the (re-)evaluation of functions altogether is certainly better than any execution strategy. This is the promise of *memoization* [113, 114]: once we have spent the effort to evaluate `f(args)` to value `res`, memoize the pair `(args, res)` and immediately respond with `res` on subsequent calls with arguments `args`. Memoization may be absolutely necessary for UDFs like `floyd`, which otherwise perform $O(3^n)$ recursive calls. In this section, we will show how to add memoization to the SQL-based interpreter of `floyd` without changing the original function. This is possible because the recursive CTE already computes the results of all recursive calls and stores them in the union table `u`. We can therefore use `u` to extract the results of previous calls and avoid re-computation.

[113]: Michie (1968), ‘Memo’ Functions and Machine Learning’

[114]: Bird (1980), ‘Tabulation Techniques for Recursive Programs’

The SQL-based interpreter can provide memoization for any UDF `f`. No change to `f` is required. To do this, we associate an n -ary UDF `f` with a table `memo(args, res)` of $n + 1$ columns (for `floyd`, this `memo` table has columns `n, s, e, res` with key `(n, s, e)`). The following lines extend `floyd`’s interpreter of Figure 9.20 to do a lookup in `memo` for the current arguments `(r.n, r.s, r.e)`:

```

8 LATERAL ((lookup res in table memo for (r.n,r.s,r.e))) AS m("memo?",res),
9 LATERAL (
10 SELECT true, 'apply', NULL, n, s, e, m.res AS x, KS
11 WHERE call = 'floyd' AND m."memo?"
12 UNION ALL

```

On a successful lookup, indicated by `m."memo?" = true`, the memoized value `m.res` is passed directly to the current continuation on the stack `KS` (Line 10). In effect, the entire subtree of recursive calls below `floyd(n, s, e)` is truncated, regardless of whether the call occurs at the top level or deep in the recursion. This can save a lot of computation effort.

How do we fill in the table `memo`? For an answer, examine the call tree for the top-level call `floyd(2,2,3)` in Figure 9.22. When the recursive call ② to `floyd(0,2,3)` has computed the intermediate

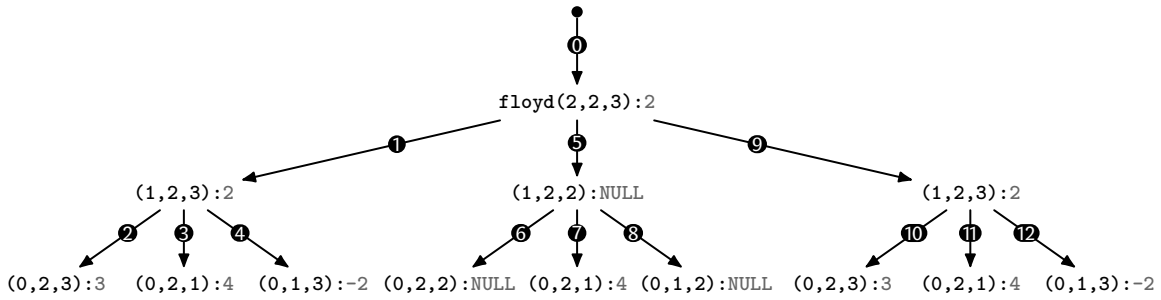


Figure 9.22.: Call tree for top-level call `floyd(2,2,3)`. Edge $\bullet \text{---} \textcircled{1} \text{---}$ indicates the i th call performed by the interpreter. Grey values denote the results of the calls.

result 3, it passes the values $x=3$ to the top continuation on the stack κ_s (which will continue with call 3 as determined by the CPS transformation). Since the union table u collects a log of all such continuation invocations in rows with $l = \text{'apply'}$ (see column x in the row annotated with call 2 in Figure 9.21), it is a viable source for `memo` entries:

- (M1) Run the CTE-based interpreter, get the union table u .
- (M2) In u , find all rows r with $r.l = \text{'apply'}$. If not already present, insert row $(args, r.x)$ into table `memo`, where `args` are the arguments of the current call. (We find 13 such rows when the interpreter has evaluated `floyd(2,2,3)`, corresponding to the 13 nodes in the call tree in Figure 9.22. The lookup in the added Line 8 above will find these entries in subsequent interpreter runs.) To facilitate this, we have carefully designed the closure records to provide `args`. In Figure 9.21, for call 2, the topmost closure records k_1 would hold the arguments $(n, s, e) = (0, 2, 3)$. Similarly, k_2 would hold $(1, 2, 3)$ for call 9.

For a graph with n nodes, `floyd`'s `memo` table will contain n^3 rows once it is completely filled. This form of memoization is highly efficient due to built-in index support for the key lookups performed by Line 8. It is important to note that the applicability of memoization depends on the UDF being referentially transparent, either in general (IMMUTABLE functions [33, §38.7]) or at least within a transaction context (STABLE functions like `floyd` due to their access to table `edges`, see Line 6 in Figure 8.1). The lifetime of the table `memo` is also defined by these degrees of referential transparency.

[33]: *POSTGRESQL 15 Documentation*

9.6. Implementation of Continuation Stacks

So far, we have kept the implementation of the continuation stack abstract, requiring only that the operations `TOP`, `EMPTY`, `TAIL`, and `PUSH` are supported. Of course, these operations must be defined on the basis of a data type supported by SQL. Since no database system supports a *real* stack data type, we have to resort to another representation. The SQL `array` type is a possible SQL-side implementation of the continuation stack in column κ_s . The `TOP`, `TAIL`,

```

TOP( $\kappa$ )  $\mapsto$   $\kappa[1]$ 
TAIL( $\kappa$ )  $\mapsto$   $\kappa[2:]$ 
PUSH( $t, \kappa$ )  $\mapsto$   $t || \kappa$ 
EMPTY( $\kappa$ )  $\mapsto$  cardinality( $\kappa$ ) = 0
EMPTY_STACK()  $\mapsto$  ARRAY []
    
```

Figure 9.23.: One possible mapping from abstract stack operations to SQL.

and PUSH operations then effectively operate on the array head element. Figure 9.23 shows how the abstract stack operations can be translated into valid SQL expressions.

While this design works and exhibits the correct behavior, the length of the array is determined by the recursion depth. If we recurse deeply, large κs entries result in measurable overhead when the CTE assembles instruction tuples to place in the working and union tables.

Continuation Stacks Outside the Working Table and Union Table.

We have experimented with a PostgreSQL extension that hosts the continuation stack outside the working and union tables. Here, the κs column simply refers to a table-like structure of closure records that live in a separate region of memory that is private to the SQL query that executes the interpreter. Chapter 10 reports on the runtime advantages of replacing the array-based stack with this tabular representation. Keep in mind, however, that such an extension requires support from the database system, and is usually not portable.

Recursive UDF processing, through repeated unfolding and planning of function bodies, makes relational DBMSs poor *programming* environments [104, 115]. We believe that it does not have to be this way: the SQL UDF compilation described in Chapter 9 can turn database systems that support modern SQL, such as PostgreSQL, into a viable functional programming platform on which complex computations can be performed *with and alongside* tabular data.

To illustrate this point, the 10 recursive UDFs in Table 6.1¹ address algorithmic problems that would not typically be considered database-resident computations due to (1) their inefficiency when expressed as SQL functions, or (2) the prohibitive complexity of manually formulating them in terms of a recursive CTE. Here, we implement them as recursive UDFs in the compact and readable functional style of `floyd` (Figure 8.1). We chose these UDFs to show a variety of recursion patterns (see the **Recursion** column in Table 6.1).

For reference, the measurements below report the average of five runs performed with PostgreSQL version 13. We rely on the vanilla system except where we explicitly mention the use of the query-private table storage extension, recall Section 9.6. The database system was hosted on a 64-bit Linux machine (two AMD EPYC™ 7402 CPUs at 2.8 GHz and 512 GB of RAM, 128 GB of which were assigned to hold the database buffer). The execution stack size of the database server was set to 6 MB, which was sufficient to hold the frames of all recursive UDFs in our experiments.

Reducing Function Call Overhead (No Memoization). Compilation into CTE form results in iterative SQL queries that no longer perform recursive UDF calls. The saved function call overhead (remember Figure 8.3) is the runtime reduction we are looking for. In fact, we find that this overhead accounts for about 95% of the total runtime of SQL queries Q that repeatedly invoke the UDFs with random arguments (averaged over all UDFs, see column **Overhead**). It is now clear that Figure 8.3 painted an optimistic picture: the *eval* phases of useful work tend to be no more than $1/20$ Q 's total time span $t_\omega - t_\alpha$.

Even without using memoization, UDF compilation reduces this overhead to about 8% on average. The remaining overhead is due to Q 's invocation of the non-recursive UDF that wraps the CTE (see Figure 9.20). If this residual overhead is noticeable—*e.g.*, for computationally lightweight functions like `fsm` and `paths`, or for frequently called UDFs (`mbrot` is called 16950 times by Q)—it may be advisable to inline the CTE at the UDF call site(s) in Q . This significant reduction in call overhead is reflected in the **Time/Call** column, which shows the average runtime per top-level function call before and after compilation. A reduction of the call time by a factor of 10 is typical. For UDF `vm`, we measured an improvement of 180 times: `vm` is structured in the form of 9 conditional branches. Each

[104]: Duta et al. (2020), ‘Functional-Style SQL UDFs with a Capital ‘F’’

[115]: Aranda et al. (2013), ‘R-SQL: An SQL Database System with Extended Recursion’

1: The full SQL definitions of these UDFs are given in Appendix B.

Table 10.1. Impact of compilation and memoization for 10 recursive SQL UDFs.

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

branch handles one type of VM instruction. While the branches are mutually exclusive, all 9 contain recursive calls to `vm`, which are unfolded (once) and then planned for each call. None of this effort remains after compilation.

Impact of Memoization. Recursive CTEs require time and space to construct the union table `u`. Our memoization approach aims to take advantage of this effort and reuse the union table. Tail-recursive functions do not require a stack. Therefore, the `mbrot`, `path`, and `vm` UDFs only memoize the top-level call. The **Memoize** column of Table 10.1 documents the runtime impact of memoization when we enable it for a sequence of 15000 calls to the compiled UDFs. Over time (from left to right), recursive calls find their random arguments in the `memo` table with increasing probability, and as expected, call times decrease. All 10 UDFs listed in Table 10.1 benefit from memoization. The behavior of `dtw` reflects our choice of arguments in this particular case: the function is evaluated over time series of increasing length, and the timings ramp up until the maximum sequence length is reached. At this point, the table `memo` has fully materialized the function [104]. Note that for some UDFs, the effects of memoization become apparent only after a large number of calls: for `comps`, timings evolve like over the course of 150000 calls.

[104]: Duta et al. (2020), ‘Functional-Style SQL UDFs with a Capital ‘F’’

Zooming in on UDFs `march` and `eval`. In the context of database applications, it is typical for a SQL query `Q` to invoke a UDF multiple times. The plots in Figures 10.1a and 10.1b show the total runtime of queries `Q` that perform between 50 and 5000 top-level invocations of the UDFs `march` and `eval`, respectively. Both plots show the order of magnitude runtime differences between UDFs (∞) and their CTE (∞) equivalent. The experiment also shows the effects of the CTE optimizations described in Sections 6.3 and 9.6.

UDF `march` uses linear recursion to implement the Marching Squares algorithm (recall Section 4.1, where we described an iterative version of the algorithm), which traces the border of an object in the 2D plane. Each step of recursion adds one point to the border, leading to recursion depths of up to 480 in our experiments (far beyond the depth limits enforced by engines like ORACLE or SQL SERVER). Thus, when `march` is compiled and evaluated as a CTE, we find array-encoded continuation stacks of the same length in the `ks` column of the working and union tables. When the CTE-based interpreter pushes on these stacks and embeds them in the next statement to execute, POSTGRES SQL performs expensive array copy operations.

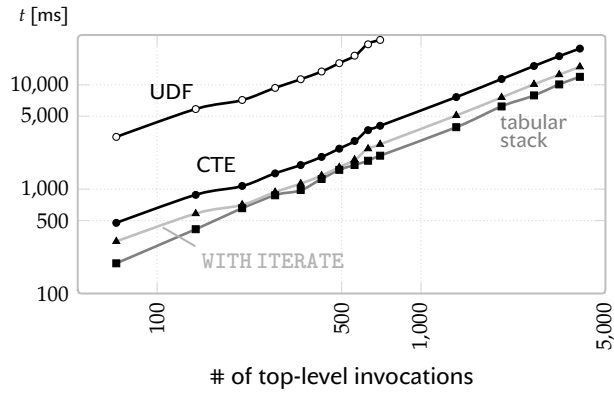
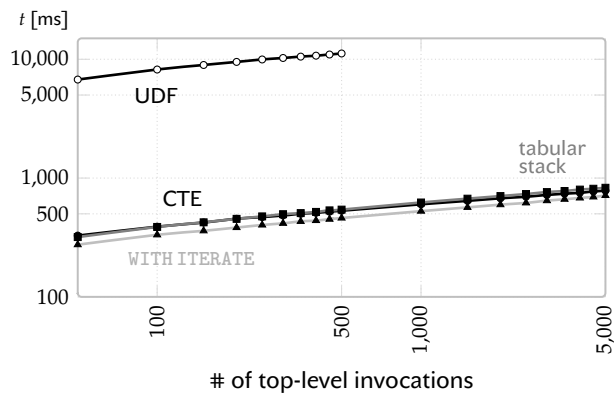
(a) UDF `march` (Marching Squares).(b) UDF `eval` (expression evaluation).

Figure 10.1. Runtime of query `Q` before/after compilation, impact of CTE optimizations.

The tabular continuation stack representation outside the working and union tables described in Section 9.6 avoids these copy costs and allows constant-time `PUSH` and `TAIL` operations. The runtime measurements `■` in Figure 10.1a manifest these savings. In addition to large stacks, `march` has to deal with the construction of a potentially large function result: ever-longer arrays of border points accumulate in the column `res` of the rows in the union table. Switching from `WITH RECURSIVE` to `WITH ITERATE` can avoid the associated row construction and table maintenance (at the cost of disabling memoization), see `▲` in Figure 10.1a.

Both optimizations have a negligible impact on `eval`. The UDF performs bottom-up evaluation of subexpressions in a large arithmetic expression tree. The tree depth of 16 defines the maximum recursion depth. This leads to short continuation stacks in column `ks`, which are handled efficiently even in their vanilla array representation: the tabular stack optimization does not pay off (`■` and `●` overlap in Figure 10.1b). Also, the CTE for `eval` keeps comparatively compact results of type `numeric` in the `res` column of the union table. The use of `WITH ITERATE` also has little effect (`▲`), and the system works just fine with the standard `WITH RECURSIVE` construct.

From Recursion to Iteration to SQL—Marching Squares

11.

Recursion is a powerful part of a programmer’s toolbox. Some problems are much easier to solve using a recursive formulation and many textbook-style algorithms are defined recursively. Iterative versions are not always available. Using recursion in the context of database systems is untypical at best, but disastrous at worst. We have already established this for recursive SQL UDFs, and now we will take a look at recursive PL/SQL UDFs.

Recursive PL/SQL UDFs suffer from the same problems during execution as recursive SQL UDFs do, as described in Chapter 8. If execution is possible at all, the ever-increasing stack size (recall Figure 8.3) may cause the system to abort the computation when the system’s available process memory is exhausted. As a result, database systems typically either restrict recursion to some arbitrary limits or do not support it at all. Some systems like PostgreSQL are able to execute such functions without major restrictions, however, as we will see in Section 11.3, the performance of naive execution is devastating.

In Chapter 9, we have extended the PL/SQL compilation chain from Part 1 to be able to handle recursive SQL UDFs. We intentionally converted the recursive SQL functions to the SSA_{REC} IR before converting back to the ANF_{REC} equivalent. This was done in preparation for the compilation of recursive PL/SQL functions. In this chapter, we will use this compilation pipeline to compile both a scalar (see Figure 11.3) and a table-valued (see Figure 11.6) recursive PL/SQL version of UDF `march` to trampolined-style SQL.

11.1. Recursive Marching Squares

Recall that the Marching Squares algorithm [66] computes the contour of a 2D object by *marching* around it (see Section 4.1). The algorithm returns the 2D coordinates of the contour as the result. The recursive SQL UDF in Figure 11.1 expresses this behavior using four black boxes. Q_1 checks whether we have reached the base case or whether we need to evaluate the recursive case. When we reach the base case, we are done and return the empty array. Otherwise, Q_4 computes the direction of the next move. Based on this result, Q_3 calculates the next position on the contour, which is used to recursively call the `march` UDF again. Finally, Q_2 combines the result of the recursive call with the current position and returns it. This evaluation order corresponds to the statement order of the SSA program in Figure 11.2 after flattening the nested black boxes using the translation rules in Section 9.1.

This formulation of the `march` UDF is considered linear recursive because the function calls itself only once, but not in tail position. The depth of the recursion is equal to the number of points on the contour of the 2D object. This can easily become a problem, as the

[66]: Maple (2003), ‘Geometric design and space planning using the marching squares and marching cube algorithms’

```

1 CREATE FUNCTION march(cur point, goal point) RETURNS point[] AS
2 $$
3 SELECT CASE
4   WHEN (v0 = v1)[cur,goal] THEN array[] :: point[]
5   ELSE (v1 || v0)
6     [ march(
7       point(v1.x + (v0.dir).x, v1.y + (v0.dir).y)
8       [, (SELECT d
9         FROM squares AS s, directions AS d
10        WHERE v0 = s.xy
11          AND (s.ll,s.lr,s.ul,s.ur)
12             = (d.ll,d.lr,d.ul,d.ur))[cur]
13         , cur ]
14       , goal)
15     , cur ]
16 END;
17 $$ LANGUAGE SQL STRICT;

```

Figure 11.1: Recursive SQL UDF march.

resulting stack frames quickly consume all of the available stack memory of the database system.

11.2. Recursive PL/SQL

```

fun march(cur, goal) : point[] {
  start:
  IF Q1[cur,goal]
  THEN RETURN [] :: array[];
  ELSE GOTO κthen

  κthen:
  dd ← Q4[cur]
  new_cur ← Q3[cur, dd]
  REC v1 = march(new_cur, goal) IN
  RETURN Q2[new_cur, v1]
}

```

Figure 11.2: SSA_{REC} representation of scalar recursive version of UDF march.

Instead of using a language SQL UDF, we can formulate the same algorithm using a recursive PL/SQL UDF. This allows us to combine imperative statements (*i.e.*, loops, conditionals, *etc.*) with recursion. We can use statement sequencing to organize the evaluation instead of using nested black boxes as in Figure 11.1. Figure 11.3 shows this version of march.

```

1 CREATE FUNCTION march(cur point, goal point) RETURNS point[] AS
2 $$
3 DECLARE
4   new_cur point;
5   dd directions;
6   v1 point[];
7 BEGIN
8   IF (cur = goal)
9   THEN RETURN [];
10  ELSE
11    dd := (SELECT d
12          FROM squares AS s, directions AS d
13          WHERE s.xy = cur
14            AND (s.ll,s.lr,s.ul,s.ur)
15               = (d.ll,d.lr,d.ul,d.ur));
16
17    -- Calculate next position on 2D contour
18    new_cur := point(cur.x + (dd.dir).x, cur.y + (dd.dir).y);
19    v1 := march(new_cur, goal);
20    RETURN new_cur || v1;
21  END IF;
22 END
23 $$ LANGUAGE PLpgSQL STRICT;

```

Figure 11.3: Recursive PL/SQL UDF march with scalar return type point[].

Although the recursive SQL version in Figure 11.1 and the recursive PL/SQL version in Figure 11.3 initially use different languages to express this algorithm, once compiled into SSA (again, see Figure 11.2), it is obvious that both versions are equivalent.

After defunctionalization, we end up with the code shown in Figure 11.4. Since the function is linear recursive, there is only one

continuation generated by the CPS transformation. This gives us some opportunities for improvement. The `CASE 1 OF` statement in function `apply` is not needed because there is only one branch to select (see Lines 14 and 15 in Figure 11.4). Therefore, we can eliminate this statement, and remove the corresponding continuation label in the continuation stack type, which simplifies the stack to type `point []`.

```

1 fun march(cur,goal,x,κs) : point[] {
2   IF Q4[cur,goal] THEN apply(cur, goal, [], κs)
3   ELSE
4     LET dd = Q4[cur] IN
5     LET new_curr = Q3[cur,dd] IN
6     march(new_curr, goal, NULL, PUSH([κ1, new_curr]), κs))
7 }
8
9 fun apply(cur,goal,x,κ) : point[] {
10  IF EMPTY(κ) THEN x
11  ELSE
12    LET ⟨l, new_cur⟩ = TOP(κ) IN
13    LET κs = TAIL(κs) IN
14    CASE 1 OF
15      'κ1': apply(cur, goal, Q2[new_cur, x], κs)
16  }

```

Figure 11.4.: Defunctionalized version of UDF `march` in Figure 11.2.

After applying the remaining compilation steps, we end up with the trampolined-style version of SQL shown in Figure 11.5. This translation is correct and computes the correct result, but it suffers from the same array maintenance and copying overhead as described in Section 4.1 that blows up the working table size. In this case, it is even worse, because both columns `res` and `k` (the continuation stack) are of type `point []`. The CPS and defunctionalization-based translation results in a two-phase trampolined style evaluation of UDF `march` in Figure 11.5:

`call = 'march'`: While descending recursively (see Lines 8 to 18), the continuation stack grows continuously (see `r.cur || r.k` in Line 17). Due to the simplification of the data type, only the current contour point `r.cur` needs to be pushed to the stack. This results in a total of $\frac{1}{2}n^2 \times (n - 1)$ copy operations.

`call = 'apply'`: During the recursive ascend (see Lines 18 to 27), the CTE effectively iterates over the continuation stack, concatenating all elements with the `res` column (see Line 25). This, in turn, results in another $n^2 \times (n - 1)$ copy operations.

The translation therefore requires a total of $\frac{3}{2}n^2 \times (n - 1)$ copy operations, which has a negative impact on performance. Ideally, an optimization step would recognize that the continuation stack already contains the final result after the recursive descent. This would completely eliminate the need to ascend from the recursion in Lines 8 to 18, reducing the number of copy operations to $\frac{1}{2}n^2 \times (n - 1)$. This would be an improvement, but still not ideal. However, we will not use such an optimization in the following, but return to the idea of Chapter 4 and consider using a TVF instead. The ideal solution would be to eliminate the need for the copy operations altogether, which TVFs allow us to do.

```

1 CREATE FUNCTION march(cur point, goal point) RETURNS point[] AS
2 $$
3 WITH RECURSIVE run("rec?", call, res, cur, goal, k) AS (
4   SELECT true, 'march', NULL :: point[], cur, goal, ARRAY[] :: point[]
5     UNION ALL
6   SELECT  .*
7   FROM    run AS r,
8   LATERAL (SELECT true, 'apply', NULL :: point, r.cur, r.goal, r.k
9            WHERE r.call = 'march' AND r.cur = r.goal Q1[?])
10          UNION ALL
11          SELECT r.*
12          FROM  (SELECT d.dir
13                  FROM squares AS s, directions AS d
14                  WHERE r.cur = s.xy
15                        AND (s.ll,s.lr,s.ul,s.ur)
16                            = (d.ll,d.lr,d.ul,d.ur)) AS d(dir),
17          LATERAL (SELECT true, 'march', NULL, point(cur.x + dir.x, cur.y + dir.y), r.goal, r.cur || r.k) AS r
18                  WHERE r.call = 'march' AND NOT (r.cur = r.goal) Q1[?])
19          UNION ALL
20          SELECT apply.*
21          FROM  (SELECT r.k[1]) AS _(current),
22          LATERAL (SELECT false, NULL, r.res, r.cur, r.goal, r.k
23                  WHERE CARDINALITY(r.k) = 0
24                  UNION ALL
25                  SELECT true, 'apply', current || r.res, current, r.goal, r.k[2:]
26                  WHERE CARDINALITY(r.k) <> 0) AS apply
27          WHERE r.call = 'apply'
28 WHERE run."rec?"
29 ) SELECT r.res FROM run AS r WHERE r."data?"
30 $$ LANGUAGE SQL strict;

```

Figure 11.5.: Final trampolined-style SQL version of the scalar recursive UDF `march` in Figures 11.1 and 11.3.

Recursive Table-Valued PL/SQL. Using recursive PL/SQL allows us to formulate a table-valued version of `march` that does not need to copy array elements at all (see Figure 11.6). While the basic structure of Figure 11.6 is identical to Figure 11.3, black box `Q2` no longer exists. This piece of logic has been replaced by the `RETURN NEXT` and `RETURN QUERY` statements in Lines 19 and 20. Since there is no further interaction with the result of the recursive call to `march` (see Line 20), the function is effectively tail-recursive and therefore identical to the table-valued iterative formulation in Figure 4.2a of Section 4.1. This eliminates the need for the continuation stack and the result array, which greatly improves performance. Figure 4.5 shows the final trampolined-style translation.

The key point to note is that even though we started with a UDF definition that was recursive, we ended up with the same end result as if we had used a loop. This is excellent, because it means that using recursion does not necessarily have to be a runtime penalty. We have defined a recursive formulation of the algorithm, but the compiler has optimized it to be as efficient as the iterative version.

```

1 CREATE FUNCTION march(cur point, goal point) RETURNS SETOF point AS
2 $$
3 DECLARE
4     new_cur point;
5     dd directions;
6 BEGIN
7     IF (cur = goal)
8     THEN RETURN;
9     ELSE
10      dd := (SELECT d
11             FROM squares AS s, directions AS d
12             WHERE s.xy = cur
13             AND (s.ll,s.lr,s.ul,s.ur)
14                = (d.ll,d.lr,d.ul,d.ur));
15
16      -- Calculate next position on 2D contour
17      new_cur := point(cur.x + (dd.dir).x, cur.y + (dd.dir).y);
18
19      RETURN NEXT cur;
20      RETURN QUERY (TABLE march(new_cur, goal));
21     END IF;
22 END
23 $$ LANGUAGE PLpgsql STRICT;

```

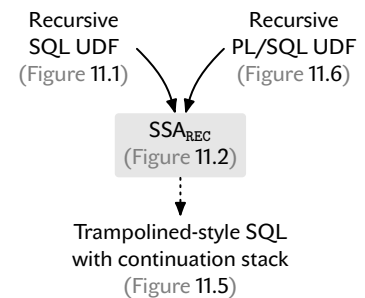
Figure 11.6.: Recursive PL/SQL UDF march with table-valued return type.

11.3. Experiments

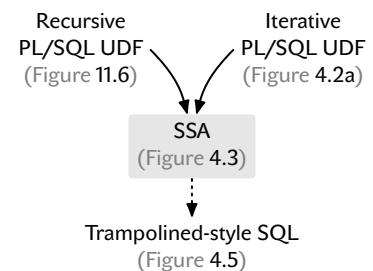
In the following, we will compare several different versions of UDF `march` described in the previous Section 11.1 and in Chapter 4. All of the measurements have been performed on a 64-bit Linux x86 host (two AMD EPYC™ 7402 CPUs at 2.8 GHz and 2 TB of RAM). If not stated otherwise, the timings were averaged over five runs, ignoring the worst and best times. All measurements used PostgreSQL version 15 with 4 GB of working memory. The execution stack size was set to 6 MB.

Speedup Over Naive SQL UDF march. In Figure 11.8 we show the speedup factors of the different scalar versions of UDF `march` over the recursive SQL UDF in Figure 11.1:

- shows the speedup factor of the recursive PL/SQL UDF in Figure 11.3 over the recursive SQL UDF in Figure 11.1. This version can only handle input sizes up to 256, even with the stack size increased to 6 MB. However, for the available data points, it is up to about 12 times faster than the recursive SQL UDF formulation. The average speedup factor is 8.25.
- compares the speedup of the iterative version of `march` as shown in Figure 4.2a of Chapter 4. Compared to the recursive PL/SQL formulation ■, this version performs slightly better, as expected. However, up to an input size of about 64, the versions perform similarly. While this version maintains a speedup factor of about 15 to 20 (12.78 times faster on average), the recursive formulation experiences a performance degradation. This is due to the allocation and deallocation of stack frames.
- shows the speedup factor of the compiled version of Figures 11.1 and 11.3, which is the trampolined-style SQL version with a continuation stack in Figure 11.5. While this version can handle all input sizes, the speedup factor peaks at an input size of 32



(a) Overview of the scalar versions of UDF `march`.



(b) Overview of the table-valued versions of UDF `march`.

Figure 11.7.: Provides an overview of the different input and translated output versions of UDF `march`.

and degrades significantly to a factor of about 2.3. This version manages to improve performance by an average factor of 5.95. The recursive PL/SQL version ■ performs slightly better than this version of the translation.

- shows the speedup factor of the translation of the iterative version of `march` as shown in Figure 4.2b. This version does not have to manage the continuation stack, which avoids many expensive copy operations. As a result, this version has the best performance, with a speedup factor of up to 35 and 22.29 on average.

Both SQL versions (■ and ■) use the regular `WITH RECURSIVE` semantics, not the optimized `WITH ITERATE`. We will investigate the effect of `WITH ITERATE` later in Figure 11.10.

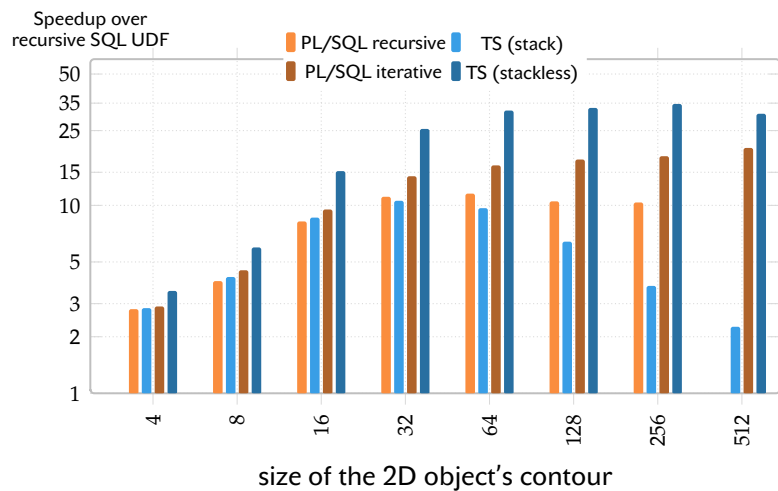


Figure 11.8: Speedup factor of the scalar UDF versions of `march` over the naive recursive SQL UDF.

Table-Valued Versions. Similar to Figure 11.8, Figure 11.9 shows the speedup factors of the different table-valued versions of UDF `march` over the recursive SQL UDF in Figure 11.1. Note that the versions also use `WITH ITERATE`:

- this bar shows the recursive PL/SQL UDF version with a table-valued return type, as shown in Figure 11.6. It performs slightly worse than its scalar counterpart, with an average speedup factor of 5.93. However, it can handle all input sizes. This function suffers from the same stack-related performance degradation at large input sizes as the non-table-valued version.
- shows the speedup factor of the iterative table-valued version of `march`, as in Figure 4.2a. This variant performs almost identically to its scalar equivalent, with an average speedup factor of 13.01. There is no performance degradation for larger inputs.
- shows the translation of ■ and ■. This version does not require an explicit continuation stack because the function is effectively tail-recursive (see Figure 11.6). Without this stack, and without an array return type, the expensive copy operations are eliminated. As a result, this version improves performance by an average factor of 25.83. Removing these array data types from the working table also eliminates performance degradation with larger input sizes.

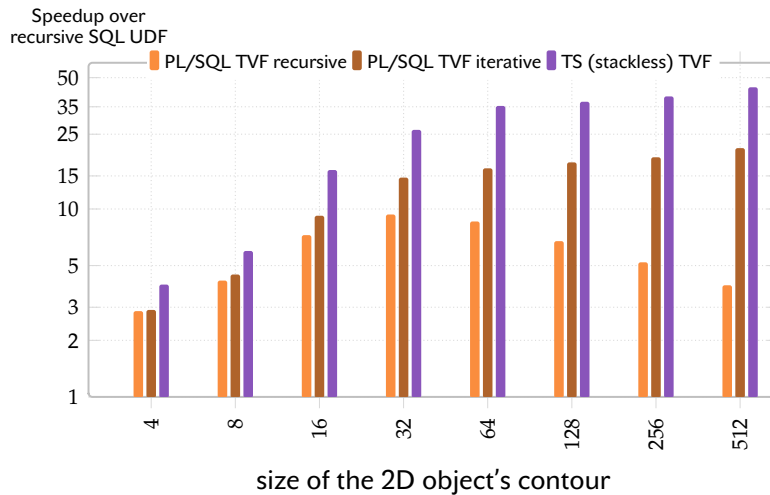


Figure 11.9.: Speedup factor of the table-valued UDF versions of march over the naive recursive SQL UDF.

The Effect of WITH ITERATE. The space-saving effect WITH ITERATE (recall Section 6.3) provides is particularly interesting in the case of recursive UDFs, since managing the continuation stack can become expensive.

- shows the speedup factor of WITH ITERATE over WITH RECURSIVE in Line 3 of Figure 11.5.
- shows the speedup factor when using WITH ITERATE instead of WITH RECURSIVE in the translation of Figure 4.2b of Chapter 4.
- shows the speedup factor when using WITH ITERATE instead of WITH RECURSIVE in Figure 4.5 of Chapter 4.

While WITH ITERATE improves performance for all translations, there is only a small benefit for versions without an explicit continuation stack. However, since the translation of Figure 11.5 requires many copy operations, WITH ITERATE really shines and reduces the memory footprint and thus performance. This effect is amplified by the size of the input. Using WITH ITERATE improves the average speedup factor of ■ in Figure 11.8 from 5.95 to 8.95. The translation is then at least equal to, and for most input sizes faster than, the original recursive PL/SQL version.

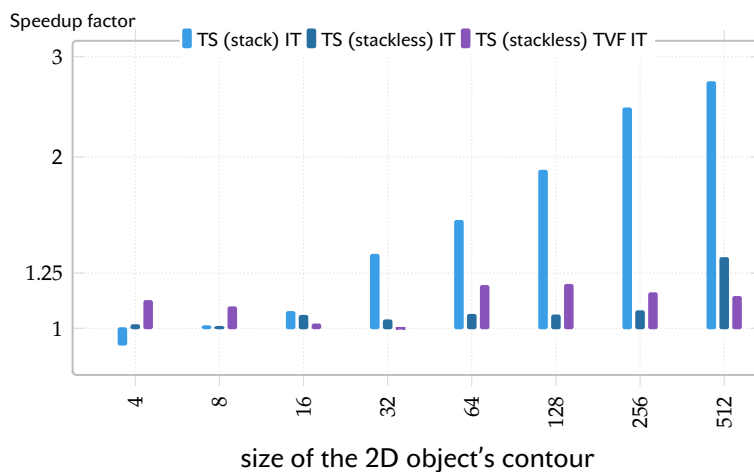


Figure 11.10.: Speedup factors of WITH ITERATE over WITH RECURSIVE.

12.1. Conclusions

We are confident that the SQL UDF compiler as described in Chapter 9 is more than just a curious excursion into the realm of programming language compilation techniques. The compiler can help to make recursive UDFs more accessible to users. This is because transforming recursive UDFs into pure SQL queries makes the computation more efficient and can be used in more database engines. The runtime savings reported for 10 recursive UDFs in Chapter 10 of about 90% are significant. The applicability is immediate, as we are pursuing a non-invasive source-to-source transformation that can be implemented on top of any database engine with SQL:1999 support. In addition to the performance benefits, the compilation approach also provides a way to write recursive UDF in a more natural way. We think this is very important because it allows more types of computation to be expressed directly in database systems. The survey done in [105] suggests that users tend to prefer recursive UDFs when the algorithm itself is recursive. This is because using a recursive UDFs is a more direct way to express the recursive algorithm.

[105]: Duta (2022), 'Another Way to Implement Complex Computations: Functional-Style SQL UDF'

The array representation of the continuation stack is the main drawback of using CPS and defunctionalization. While this data type is supported by most database engines, it is not efficient to use within recursive CTEs. This is because the array must be copied for each iteration of the recursive CTE. This is a major performance bottleneck. We have shown that the performance of recursive UDFs can be improved by using `WITH ITERATE` instead of `WITH RECURSIVE`. However, this is not a general solution because `WITH ITERATE` is not standards-compliant and is therefore not supported by database engines.

Intra-function Call Memoization. The memoization technique described in Section 9.5 depends on the entries in the `memo` table. However, these entries are not added to the table until after the UDF call is completed. This prevents the function from reusing already computed values during a single call, thus limiting the impact of memoization. There are two ways to solve this problem:

- (1) We could add another array to the columns of the working table that temporarily holds the memoization rows. This array would allow us to use memoization already during the current function call. While this approach can improve the performance of functions with many overlapping subproblems, it can actually slow down performance in cases where there are fewer subproblems. For few overlapping subproblems, the memoization rows may introduce excessive array copy overhead. But a compiler could decide whether to add such an

array based on the shape of the recursion. For example, tail recursions and linear recursions do not benefit from such a data structure, because a reappearing subproblem would mean that the program is in an infinite loop. However, if the recursion is tree-shaped, then this data structure can be used to improve performance. We have not implemented this idea, but we believe that it is a promising approach that could improve the performance of recursive UDFs. We leave this as future work.

- (2) We could try to execute the function calls in such a way that the subproblems are solved first. This would fill the `memo` table iteratively, generating solutions to larger and larger subproblems by using the solutions to smaller subproblems. We used this idea for UDF `dtw` in Chapter 10, Table 10.1.

However, in order to automate this process, it is necessary to specify the order in which the evaluations should be performed. This does not fit well with our CPS-based approach, because the CPS transformation linearizes the execution of function calls and defines an explicit evaluation order. But the function calls to be executed next are not known at compile time. We would have to generate a call graph that we could evaluate from the bottom up. A similar idea has been explored in the context of functional-style SQL UDFs by Duta and Grust [104, 105, 116].

[104]: Duta et al. (2020), ‘Functional-Style SQL UDFs with a Capital ‘F’’

[105]: Duta (2022), ‘Another Way to Implement Complex Computations: Functional-Style SQL UDF’

[116]: Duta (2022), ‘Viability of Recursive SQL Functions’

12.2. Related Work: Functional-Style SQL UDFs

Duta and Grust [104, 105, 116] also propose a strategy for compiling recursive SQL UDFs into pure SQL queries. This compiler, too, generates a recursive CTE that can replace the body of a recursive UDF. However, the compilation approach and the resulting function evaluation approach are different. After compilation, the former recursive SQL UDFs are evaluated in two distinct phases. First, a call graph is constructed as an explicit tabular data structure that records the arguments of all recursive calls that a recursive function, say `f`, would make. These calls are not yet evaluated. Second, the call graph is traversed from the bottom up, evaluating the body of `f` for the recorded calls. When the root of the call graph is reached, the final result is returned.

Using a call graph allows them to apply several optimizations, such as call sharing, which can reduce the size of the call graph, or memoization, which allows them to reuse the results of already evaluated recursive calls. However, the approach in its current state is limited to recursive SQL UDFs only. Compilation requires a slicing step that extracts only the relevant parts of the function body definition needed for the recursive call. Program slicing [117, 118] is well understood in general, but not for SQL queries.

[117]: Tip (1994), ‘A survey of program slicing techniques’

[118]: Weiser (1984), ‘Program Slicing’

12.3. Related Work: First-Class Functions for First-Order Database Engines

Grust, Schweinsberg, and Ulrich [119, 120] describe a defunctionalization approach for queries and show how this technique can be applied to query languages (*e.g.*, PL/SQL). They recognize that functions can be interpreted as values, which allows higher-order functions to be expressed in terms of regular values. Their goal is to bring functional programming paradigms to PL/SQL, without having to change existing database systems. To do that, they define a higher-level version of PL/SQL as their input language, which they then successively translate into regular PL/SQL. Similar to our approach, closures (in our case defunctionalized continuations) are represented as row values. A dispatcher implements dynamic function calls for closures. The dispatching logic itself is implemented in PL/SQL. This allows them to use the existing PL/SQL runtime to evaluate the function body.

In terms of performance, they found that dispatched function calls had minimal impact on ORACLE's performance compared to regular static calls. For POSTGRESQL 9.2, however, it caused a significant slowdown (factor 3.5). Although the slowdown would probably not be as bad with today's version of POSTGRESQL, compiling to SQL could still improve the performance of such programs. Beyond performance, higher-order functions can lead to concise and elegant formulations of algorithms that are close to the data. This is very much in line with the goals of this thesis.

While it may be possible to extend their approach to use recursive CTEs instead of PL/SQL to implement the dispatcher, this was not the focus of their work. They also do not discuss the impact of their approach on the performance of recursive UDFs.

12.4. Related Work: Fun SQL

FUNSQL [121] was proposed as a database language in the style of functional programming. This language attempts to allow developers to express application logic in imperative-like steps. While FUNSQL's function definitions are similar to PL/SQL, there are some important differences: (1) Variable assignments are in SSA and can be table-valued. (2) Tail-recursive functions are a dedicated feature. (3) Functions are compiled into a graph-based execution plan. This allows intermediate results to be read by multiple statements, and it allows cycles which is used for tail-recursion. These execution plans are compiled into sequences of SQL statements that materialize their intermediate results into temporary tables as needed. Supporting only tail recursive functions is a wise design decision because, as we have discussed, they map directly to iterations. This makes compilation much easier, and saves them the trouble of doing things like CPS and defunctionalization.

[119]: Grust et al. (2013), 'Functions are data too: defunctionalization for PL/SQL'

[120]: Grust et al. (2013), 'First-class functions for first-order database engines'

[121]: Binnig et al. (2012), 'FunSQL: It is Time to Make SQL Functional'

12.5. Incrementalization

[122]: Liu et al. (1999), ‘From recursion to iteration: what are the optimizations?’

[123]: Liu et al. (1998), ‘Static caching for incremental computation’

[124]: Liu et al. (1995), ‘Systematic derivation of incremental programs’

[125]: Bellman (1954), ‘The theory of dynamic programming’

Recursion elimination has been studied extensively. However, it has remained extremely difficult to develop a general method for transforming general recursion into iteration. Liu and Stoller [122–124] propose a powerful and systematic method to transform general recursion into iteration based on *incrementalization*. Incremental programs try to avoid unnecessary duplication of common computations by taking advantage of repeated computations on similar inputs. Given a program f and a certain input change \oplus , a program f' that efficiently computes the value of $f(x \oplus y)$ using the value of $f(x)$ is called an incremental version of f under \oplus [124].

The method described in [122] consists of three steps: (1) identify an input increment, (2) derive an incremental version under the input increment, and (3) form an iterative computation using the incremental version. The resulting programs reuse as much of the existing solution as possible. This is related to the use of memoization or dynamic programming [125]. These techniques are useful optimizations for problems that can be decomposed into overlapping subproblems. The key idea is to solve each subproblem only once and store or reuse the solution. When the same subproblem is encountered again, its solution can simply be reused rather than recomputed. The relationship between dynamic programming and incrementalization is that dynamic programming stores the solution to subproblems, while incrementalization focuses on efficiently updating the solution.

The basic premise of compiling recursive computation into iteration of the incrementalization line of work is the same as our approach described in Chapter 8 of this thesis. However, incrementalization avoids the use of CPS and defunctionalization, but more importantly, it avoids the use of a continuation stack in many cases. As we saw in Section 11.3 of Chapter 11, the continuation stack can be a major issue when compiled to SQL. Eliminating the stack could bring huge performance improvements, especially for programs with a large number of copy operations.

FINAL REMARKS

The defining goal of this thesis was to follow Rowe and Stonebraker’s advice to “move your computation close to the data”—in our case UDF-based workloads. As recognized by Ramachandra et al. [27], iterative UDFs suffer from an inherent impedance mismatch between the database system and the PL/SQL interpreter. This impedance mismatch is the root cause of the performance problems of UDFs. Starting where `FROM` and `AGGIFY` left off, we explored how to eliminate this impedance mismatch using tried and tested techniques from the field of programming languages. Most of these techniques have not been used in the context of database systems before, but we have shown how they can be adapted to work in this context.

The key insight is the SQL-based version of trampolined style, which has proven itself to be a powerful tool for expressing any kind of computation in SQL. This simple style of control flow can easily be expressed using recursive CTEs in SQL. The two paradigms complement each other and allow for a simple and efficient implementation of trampolined style in SQL. We have shown that this idea can be used to map both iterative and recursive formulations of algorithms to pure SQL queries, and that these queries can improve performance over the initial UDF formulation.

Our approach can be used on systems with UDF support to improve performance, memory usage, and the planner’s optimization capabilities. This is particularly useful in the context of legacy systems where the database system cannot be changed, but the application can be modified. In this case, the application can replace the UDF with trampolined style queries, leaving the database system untouched. But it can also be used to bring UDF support to systems without it, eliminating the need to implement, *e.g.*, a PL/SQL interpreter. For closed-source systems, this may be the only way to add UDF support. But this approach also has advantages for open-source systems. Instead of implementing a full PL/SQL interpreter, the database system only needs to support `LATERAL` joins and recursive CTEs, which is a much simpler task and already supported by many systems. The UDF compilation can then be done without modifying the database system, *e.g.*, by doing the translation in an extension if that is supported, or by doing the translation outside the database system.

The fact that we were able to share large parts of the compilation pipeline for iterative PL/SQL UDFs and for recursive UDFs is a significant accomplishment. It shows that the compilation pipeline is flexible enough to support different programming paradigms. But most importantly, it shows that trampolined style is a viable vehicle for implementing arbitrary computations in SQL. We believe that this is a significant contribution to the field of database systems in general, to the field of UDF compilation specifically, and that it will be useful for future research. We hope this work will inspire others

“Again, you can’t connect the dots looking forward; you can only connect them looking backward. So you have to trust that the dots will somehow connect in your future.”

—Steve Jobs

[27]: Ramachandra et al. (2018), ‘Froid: Optimization of Imperative Programs in a Relational Database’

to explore the possibilities of trampolined style in SQL. We are confident that there are many more applications that can be approached with this idea, because we have only scratched the surface.

13.1. Future Work

LATERAL Join Free Translation. Our method of translating UDFs to trampolined style SQL queries requires the use of `LATERAL` joins to express variable bindings. While `LATERAL` joins work fine on systems such as `POSTGRES`, some systems do not support them, or they may come with runtime penalties. Some modern database systems such as `DUCKDB` [68] and `UMBRA` [88] use a technique to unnest arbitrary correlated queries pioneered by Neumann and Kemper [126]. This technique generally improves the performance of correlated subqueries and `LATERAL` joins. However, decorrelation comes at a cost and can degrade performance when many `LATERAL` joins are used with few rows. This is exactly what our translation approach generates, unfortunately.

While our approach can be used to bring UDF computation to these modern systems, we believe that a different compilation strategy might be beneficial. It is possible to translate UDFs to trampolined style SQL queries without using `LATERAL` joins. This can be done by using a different translation scheme. Instead of translating variable bindings to `LATERAL` joins, we can translate them to non-recursive CTEs. We are confident that this dataflow-oriented translation (1) simplifies the compilation pipeline, while (2) maintaining the performance improvement on legacy systems, but (3) improves performance on modern database systems. This is something we plan to look into in the future.

Batched Evaluation. We refer to batched evaluation as the process of encoding multiple calls to the same UDF into the initial working table of the recursive CTE. This is possible because the individual control rows are independent of each other. The trampolined-style translation preserves this property, and thus allows for batched evaluation.

Huber has shown that batched evaluation has limited advantages for `POSTGRES` [127]. While some of the UDFs we have used in our experimental evaluation in Chapter 6 did benefit slightly from batched evaluation and improved performance, the benefits were not significant. To make matters worse, in some cases, batched evaluation leads to performance degradations. This result is a bit disappointing and counterintuitive because database systems promise to evaluate entire data sets more efficiently than individual rows. However, this does not seem to be the case for this particular workload. We believe this is the case because the size of the working table quickly grows larger than the available `work_mem` of the system. `POSTGRES` has no choice but to materialize pages to disk after this point. The overhead of writing to the disk was too high, even though the server

[68]: Raasveldt et al. (2019), ‘DuckDB: an Embeddable Analytical Database’

[88]: Neumann et al. (2020), ‘Umbra: A Disk-Based System with In-Memory Performance.’

`UMBRA` supports UDFs, but at the time of writing using UDFs limits the amount of parallelism the system can take advantage of.

[127]: Huber (2022), ‘Optimization of PL/pgSQL Translations Using Batching and Multiple Recursive References’

used for the measurements had a fast SSD. Also, PostgreSQL does not have any parallelization capabilities for recursive CTEs.

We think that batched evaluation can be useful for modern database systems, especially those that support parallel recursive CTEs. Promising candidates are DuckDB [68] and UmbraDB [88], which both have shown to be very fast for recursive CTEs and support parallel evaluation. We plan to investigate this in the future.

Code Generating Database Systems. Older database systems typically execute queries by interpreting them. These systems do not generate machine code for the queries. Compilers have historically not been fast enough to generate code at runtime, and the performance overhead of interpreting queries has been negligible compared to the cost of disk access [128]. Both of these reasons are no longer valid. Modern compilers are fast enough to generate code at runtime, and the performance overhead of interpreting queries is no longer negligible.

HyPer [95] and UmbraDB [88] are examples of database systems that generate machine code for queries at runtime. Also, both systems have support for UDFs similar to PL/SQL. It would be interesting to see if these compiled PL/SQL-like languages show the same performance improvements with our approach on these systems as we have seen on legacy systems, or if the code generation capabilities of these systems are able to eliminate the overhead associated with UDF execution.

Further Optimizations. We have shown that trampolined style can be used to improve performance of UDFs on legacy systems. However, there is still room for improvement. The generated SQL queries can be further optimized by the database system itself. This can be done by improving the optimizer of the system to recognize patterns generated by our translation approach and optimize them away. It may also be possible to add specialized query operators to the system that are tailored to trampoline-style queries. We think this is an interesting direction for future research.

[68]: Raasveldt et al. (2019), ‘DuckDB: an Embeddable Analytical Database’

[88]: Neumann et al. (2020), ‘Umbra: A Disk-Based System with In-Memory Performance.’

[128]: Kersten et al. (2021), ‘Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra’

[95]: Kemper et al. (2011), ‘HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots’

APPENDIX

PL/SQL UDF Definitions

A.

In the following we list the iterative PL/SQL UDF definitions of the UDFs in Table 6.1. The definitions are given in PL/pgSQL syntax.

All UDFs are available with the necessary setup code here: <https://github.com/One-WITH-RECURSIVE-is-Worth-Many-GOTOs/Code>

Function bbox.

```
1 CREATE FUNCTION bbox(start vec2) RETURNS box AS
2 $$
3 DECLARE
4     "track?" boolean := false;
5     goal     vec2;
6     bbox     box := NULL;
7     current  vec2 := start;
8     square   squares;
9     dir      directions;
10 BEGIN
11     WHILE true LOOP
12         IF "track?" AND current = goal THEN
13             EXIT;
14             END IF;
15
16         square := (SELECT s FROM squares AS s WHERE s.xy = current);
17         dir := (SELECT d
18                FROM directions AS d
19                WHERE (square.ll, square.lr, square.ul, square.ur)
20                    = (d.ll, d.lr, d.ul, d.ur));
21
22         IF NOT "track?" AND dir."track?" THEN
23             "track?" := true;
24             goal     := current;
25             bbox     := box(point(goal.x, goal.y));
26             END IF;
27         IF "track?" THEN
28             bbox := bound_box(bbox, box(point(current.x, current.y)));
29             END IF;
30
31         current := (current.x + (dir.dir).x, current.y + (dir.dir).y) :: vec2;
32     END LOOP;
33     RETURN bbox;
34 END;
35 $$ LANGUAGE PLPGSQL STRICT;
```

Function global.

```
1 CREATE FUNCTION global(orderkey int) RETURNS boolean AS
2 $$
3 DECLARE
4     regions int[];
5     region int;
6 BEGIN
7     regions := (SELECT array_agg(n.n_regionkey)
8                FROM lineitem AS l, supplier AS s, nation AS n
9                WHERE l.l_orderkey = orderkey
10                 AND l.l_suppkey = s.s_suppkey
11                 AND s.s_nationkey = n.n_nationkey);
12
13     FOREACH region IN ARRAY regions LOOP
14         IF region <> regions[1] THEN RETURN true; END IF;
15     END LOOP;
16
17     RETURN false;
18 END;
19 $$ LANGUAGE PLPGSQL;
```

Function force.

```

1 CREATE FUNCTION force(body bodies, theta float) RETURNS point AS
2 $$
3 DECLARE
4     force    point := point(0,0);
5     G CONSTANT float := 6.67e-11;
6     Q        barneshut[];
7     node     barneshut;
8     children barneshut[];
9     dist     float;
10    dir      point;
11    grav     point;
12 BEGIN
13     node = (SELECT b FROM barneshut AS b WHERE b.node = 0);
14     Q = array[node];
15
16     WHILE cardinality(Q) > 0 LOOP
17         node = Q[1];
18         Q = Q[2:];
19         dist = GREATEST(node.center<->body.pos, 1e-10);
20         dir = node.center - body.pos;
21         grav = point(0,0);
22         IF NOT EXISTS (SELECT 1
23                        FROM walls AS w
24                        WHERE (body.pos <= body.pos ## w.wall) <>
25                               (node.center <= node.center ## w.wall)) THEN
26             grav = (G * body.mass * node.mass / dist^2) * dir;
27         END IF;
28         IF (node.node IS NULL) OR (width(node.bbox) / dist < theta) THEN
29             force = force + grav;
30         ELSE
31             children = (SELECT array_agg(b) FROM barneshut AS b WHERE b.parent = node.node);
32             Q = Q || children;
33         END IF;
34     END LOOP;
35
36     RETURN force;
37 END;
38 $$ LANGUAGE PLPGSQL STABLE STRICT;

```

Function items.

```

1 CREATE FUNCTION items(catid INT8) RETURNS INT8 AS
2 $$
3 DECLARE
4     totalcount INT8;
5     curcat INT8;
6     catitems INT8;
7     subcat INT8;
8     stack INT8[];
9     catrec category;
10 BEGIN
11     totalcount := 0 :: INT8;
12     stack := ARRAY[catid];
13
14     WHILE cardinality(stack) > 0 LOOP
15         curcat := stack[1];
16         stack := stack[2:];
17         catitems := (SELECT count(P_PARTKEY) FROM item WHERE category_id = curcat);
18         totalcount := totalcount + catitems;
19         stack := (SELECT array_agg(category_id :: INT8) FROM category WHERE parent_category = curcat) || stack;
20     END LOOP;
21     RETURN totalcount;
22 END
23 $$ LANGUAGE PLPGSQL;

```


Function late.

```

1 CREATE FUNCTION late(suppkey int, orderkey int) RETURNS boolean AS
2 $$
3 DECLARE
4     lis lineitem[];
5     li lineitem;
6     blame boolean := false; -- is suppkey to blame?
7     multi boolean := false; -- does this order have multiple suppliers?
8 BEGIN
9     lis := (SELECT array_agg(l) FROM lineitem AS l WHERE l.l_orderkey = orderkey);
10    FOREACH li IN ARRAY lis LOOP
11        multi := multi OR li.l_suppkey <> suppkey;
12        IF li.l_receiptdate > li.l_commitdate THEN
13            IF li.l_suppkey <> suppkey
14                THEN RETURN false;
15            ELSE blame := true;
16        END IF;
17    END LOOP;
18    RETURN multi AND blame;
19 END;
20 $$ LANGUAGE PLPGSQL;

```

Function margin.

```

1 CREATE FUNCTION margin(partkey int) RETURNS trade AS
2 $$
3 DECLARE
4     this_order dated_order;
5     buy int := NULL; sell int := NULL;
6     margin numeric(15,2) := NULL; cheapest numeric(15,2) := NULL;
7     cheapest_order int;
8     price numeric(15,2); profit numeric(15,2);
9 BEGIN
10    -- ❶ first order for the given part
11    this_order := (SELECT (o.o_orderkey, o.o_orderdate) :: dated_order
12                    FROM lineitem AS l, orders AS o
13                    WHERE l.l_orderkey = o.o_orderkey AND l.l_partkey = partkey
14                    ORDER BY o.o_orderdate
15                    LIMIT 1);
16
17    -- hunt for the best margin while there are more orders to consider
18    WHILE this_order IS NOT NULL LOOP
19        -- ❷ price of part in this order
20        price := (SELECT MIN(l.l_extendedprice * (1 - l.l_discount) * (1 + l.l_tax))
21                FROM lineitem AS l
22                WHERE l.l_orderkey = this_order.orderkey AND l.l_partkey = partkey);
23
24        -- if this the new cheapest price, remember it
25        cheapest := COALESCE(cheapest, price);
26        IF price <= cheapest THEN
27            cheapest := price;
28            cheapest_order := this_order.orderkey;
29        END IF;
30        -- compute current obtainable margin
31        profit := price - cheapest;
32        margin := COALESCE(margin, profit);
33        IF profit >= margin THEN
34            buy := cheapest_order;
35            sell := this_order.orderkey;
36            margin := profit;
37        END IF;
38
39        -- ❸ find next order (if any) that traded the part
40        this_order := (SELECT (o.o_orderkey, o.o_orderdate) :: dated_order
41                        FROM lineitem AS l, orders AS o
42                        WHERE l.l_orderkey = o.o_orderkey AND l.l_partkey = partkey
43                        AND o.o_orderdate > this_order.orderdate
44                        ORDER BY o.o_orderdate
45                        LIMIT 1);
46    END LOOP;
47    RETURN (buy, sell, margin) :: trade;
48 END;
49 $$
50 LANGUAGE PLPGSQL;

```

Function markov.

```

1 CREATE FUNCTION markov(start_state int, success_at int, failure_at int, max_steps int)
2 RETURNS int AS $$
3 DECLARE
4   total_reward int = 0;
5   curr_state int = start_state;
6   curr_action text = '';
7   roll double precision;
8 BEGIN
9   FOR steps in 1..max_steps LOOP
10    -- Find the action the policy finds appropriate in the current state
11    curr_action = (SELECT p.action_name
12                  FROM   policy AS p, states AS s
13                  WHERE  curr_state = s.id AND p.state_id = s.id);
14    -- Random number (double precision) roll ∈ [0.0, 1.0)
15    roll = random();
16    -- Find the state we actually reach. There may be a chance we end up in another state.
17    curr_state = (SELECT possible_move.s_to
18                  FROM (SELECT a.s_to, COALESCE(SUM(a.p) OVER w, 0.0) AS p_from,
19                          SUM(a.p) OVER (ORDER BY a.id) AS p_to
20                          FROM actions AS a
21                          WHERE curr_state = a.s_from AND curr_action = a.name
22                          WINDOW w AS (ORDER BY a.id ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING)
23                          ) AS possible_move(s_to, p_from, p_to)
24                  WHERE possible_move.p_from <= roll AND roll < possible_move.p_to);
25    -- Add the reward we receive by stepping on the state we actually reached
26    total_reward = total_reward + (SELECT s.r FROM states AS s WHERE curr_state = s.id);
27    IF total_reward >= success_at OR total_reward <= failure_at THEN
28      RETURN steps * sign(total_reward);
29    END IF;
30  END LOOP;
31  RETURN 0;
32 END
33 $$ LANGUAGE PLPGSQL;

```

Function packing.

```

1 CREATE FUNCTION packing(orderkey int, capacity int) RETURNS linenum[] AS
2 $$
3 DECLARE
4     n         int;           -- # of lineitems in order
5     items    int;           -- set of lineitems still to pack
6     size     int;           -- current pack size
7     subset   int;           -- current subset of lineitems considered for a pack
8     max_size int;           -- maximum pack size found so far
9     max_subset int;         -- pack subset of maximum size found so far
10    pack     linenum[];     -- current pack
11    packs    linenum[];     -- current pack of packs
12 BEGIN
13     -- # of lineitems in order
14     n := (SELECT COUNT(*) FROM lineitem AS l WHERE l.l_orderkey = orderkey);
15     -- order key not found?
16     IF n = 0 THEN
17         RETURN array[] :: int[];
18     END IF;
19
20     -- container capacity sufficient to hold largest part?
21     IF capacity < (SELECT MAX(p.p_size)
22                   FROM lineitem AS l, part AS p
23                   WHERE l.l_orderkey = orderkey
24                         AND l.l_partkey = p.p_partkey) THEN
25         RETURN array[] :: int[];
26     END IF;
27
28     -- initialize empty pack of packs
29     packs := array[] :: linenum[];
30     -- create full set of linenumbers {1,2,...,n}
31     items := 2^n - 1;
32
33     -- as long as there are still lineitems to pack...
34     WHILE items <> 0 LOOP
35         max_size := 0;
36         max_subset := 0; -- φ
37         -- iterate through all non-empty subsets of items
38         subset := items & ~items;
39         LOOP
40             -- find size of current lineitem subset o
41             size := (SELECT SUM(p.p_size)
42                   FROM lineitem AS l, part AS p
43                   WHERE l.l_orderkey = orderkey
44                         AND subset & (1 << l.l_linenum - 1) <> 0
45                         AND l.l_partkey = p.p_partkey);
46
47             if size <= capacity AND size > max_size THEN
48                 max_size := size;
49                 max_subset := subset;
50             END IF;
51             -- exit if iterated through all lineitem subsets ...
52             IF subset = items THEN
53                 EXIT;
54             ELSE
55                 -- ... else, consider next lineitem subset
56                 subset := items & (subset - items);
57             END IF;
58         END LOOP;
59
60         -- convert bit set max_subset into set of linenumbers
61         pack := array[] :: linenum[];
62         FOR linenum IN 1..n LOOP
63             IF max_subset & (1 << linenum - 1) <> 0 THEN
64                 pack := pack || linenum :: linenum;
65             ELSE
66                 pack := pack || 0 :: linenum; -- 0 ≡ lineitem not in set
67             END IF;
68         END LOOP;
69         -- add pack to current packing
70         packs := packs || array[pack];
71
72         -- we've selected lineitems in set max_subset,
73         -- update items to remove these lineitems
74         items := items & ~max_subset;
75     END LOOP;
76
77     RETURN packs;
78 END;
79 $$
80 LANGUAGE PLPGSQL;

```

Function savings.

```

1 CREATE FUNCTION savings(orderkey int) RETURNS savings AS
2 $$
3 DECLARE
4     "order"          orders;
5     items            int;
6     lineitem        lineitem;
7     partsupp        partsupp;
8     min_supplycost  numeric(15,2);
9     new_supplier    int;
10    new_suppliers    supplier_change[];
11    total_supplycost numeric(15,2);
12    new_supplycost   numeric(15,2);
13 BEGIN
14     "order" := (SELECT o
15                FROM   orders AS o
16                WHERE  o.o_orderkey = orderkey);
17     IF "order" IS NULL THEN
18         RETURN NULL;
19     END IF;
20
21     -- # of lineitems (= parts) in order
22     items := (SELECT COUNT(*)
23              FROM   lineitem AS l
24              WHERE  l.l_orderkey = orderkey);
25
26     total_supplycost := 0.0;
27     new_supplycost   := 0.0;
28     new_suppliers    := array[] :: supplier_change[];
29
30     -- iterate over all lineitems in order
31     FOR item IN 1..items LOOP
32         -- pick current lineitem in order
33         lineitem := (SELECT l
34                    FROM   lineitem AS l
35                    WHERE  l.l_orderkey = orderkey AND l.l_linenumber = item);
36         -- find current supplier for lineitem's part
37         partsupp := (SELECT ps
38                    FROM   partsupp AS ps
39                    WHERE  lineitem.l_partkey = ps.ps_partkey AND lineitem.l_suppkey = ps.ps_suppkey);
40
41         -- find minimum supplycost (for ANY supplier that has sufficient stock) for the lineitem's part
42         min_supplycost := (SELECT MIN(ps.ps_supplycost)
43                          FROM   partsupp AS ps
44                          WHERE  ps.ps_partkey = lineitem.l_partkey
45                          AND    ps.ps_availqty >= lineitem.l_quantity);
46
47         -- new supplier with minimum supplycost
48         new_supplier := (SELECT MIN(ps.ps_suppkey)
49                        FROM   partsupp AS ps
50                        WHERE  ps.ps_supplycost = min_supplycost
51                        AND    ps.ps_partkey = lineitem.l_partkey);
52
53         -- record whether supplier has changed (part, old, new)
54         IF new_supplier <> partsupp.ps_suppkey THEN
55             new_suppliers := (lineitem.l_partkey, partsupp.ps_suppkey, new_supplier) :: supplier_change || new_suppliers;
56         END IF;
57
58         -- total supplycost of original and new supplier
59         total_supplycost := total_supplycost + partsupp.ps_supplycost * lineitem.l_quantity;
60         new_supplycost   := new_supplycost   + min_supplycost * lineitem.l_quantity;
61     END LOOP;
62
63     RETURN ((1.0 - new_supplycost / total_supplycost) * 100.0, new_suppliers) :: savings;
64 END;
65 $$
66 LANGUAGE PLPGSQL;

```

Function sched.

```

1 CREATE FUNCTION sched(orderkey int) RETURNS scheduled[] AS
2 $$
3 DECLARE
4     "order"      orders;
5     details      order_details;
6     schedule_start date;      -- production of order must happen
7     schedule_end   date;      -- between these dates
8     schedule      scheduled[]; -- constructed schedule
9     busy           daterange[]; -- when are we busy already?
10    lineitem       lineitem;   -- current lineitem to schedule
11    item_start     date;        -- production of current lineitem
12    item_end       date;        -- happens between these dates
13 BEGIN
14     -- access order, bail out if order does not exist
15     "order" := (SELECT o FROM orders AS o WHERE o.o_orderkey = orderkey);
16     IF "order" IS NULL THEN RETURN NULL; END IF;
17     details := (SELECT (COUNT(*), MAX(l.l_shipdate)) :: order_details FROM lineitem AS l WHERE l.l_orderkey = orderkey);
18     -- lineitems need to be produced between these dates
19     schedule_end := details.last_shipdate;
20     schedule_start := "order".o_orderdate;
21     schedule := array[] :: scheduled[]; -- start with an empty schedule
22     busy := array[] :: daterange[]; -- we're not busy yet
23     FOR prio IN 1..details.items LOOP
24         -- grab lineitem with given priority (~ l_extendedprice)
25         lineitem := (SELECT l.lineitem
26                     FROM (SELECT ROW_NUMBER() OVER (ORDER BY p.p_retailprice DESC) AS priority, l AS lineitem
27                          FROM lineitem AS l, part AS p
28                          WHERE l.l_orderkey = orderkey AND l.l_partkey = p.p_partkey) AS l(priority, lineitem)
29                     WHERE l.priority = prio);
30
31         -- initially, try to produce lineitem as late as possible
32         item_end := LEAST(lineitem.l_shipdate, schedule_end);
33         item_start := item_end - lineitem.l_quantity :: int;
34         -- move production forward until we find a non-busy period or
35         -- we learn that we cannot schedule the item in the available date range :-/
36         WHILE daterange(item_start, item_end) && ANY(busy) AND item_start >= schedule_start LOOP
37             item_end := (SELECT lower(b)
38                         FROM unnest(busy) AS b
39                         WHERE daterange(item_start, item_end) && b
40                         ORDER BY b
41                         LIMIT 1);
42             item_start := item_end - lineitem.l_quantity :: int;
43         END LOOP;
44         IF item_start >= schedule_start THEN
45             -- succeeded to schedule
46             schedule := schedule || (lineitem.l_linenum, item_start) :: scheduled;
47             busy := busy || daterange(item_start, item_end);
48         END IF;
49     END LOOP;
50     -- order schedule by item start date
51     IF cardinality(schedule) > 0 THEN
52         schedule := (SELECT array_agg(s ORDER BY s."when") FROM unnest(schedule) AS s);
53     END IF;
54     RETURN schedule;
55 END;
56 $$ LANGUAGE PLPGSQL;

```

Function service.

```

1 CREATE FUNCTION service(custkey int) RETURNS text AS
2 $$
3 DECLARE
4     totalbusiness float; level text;
5 BEGIN
6     totalbusiness := (SELECT SUM(o.o_totalprice) FROM orders AS o WHERE o.o_custkey = custkey);
7
8     IF totalbusiness > 1000000 THEN level := 'Platinum';
9     ELSIF totalbusiness > 500000 THEN level := 'Gold';
10    ELSE level := 'Regular';
11    END IF;
12    RETURN level;
13 END;
14 $$ LANGUAGE PLPGSQL;

```

Function sheet.

```

1 CREATE FUNCTION eval_cell(c cell) RETURNS float AS
2 $$
3 DECLARE
4     deps cell[]; cells cell[]; open jsonb[]; formulae jsonb[]; rpn jsonb[]; exprs jsonb[];
5     expr jsonb; e jsonb; root jsonb; args arguments; dep cell; intermediates contents[];
6     stack float[];
7 BEGIN
8     -- 1 compute ordered array of dependencies for cell c
9     deps := array[c]; expr := (SELECT s.formula FROM sheet AS s WHERE s.cell = c);
10    open := array[expr];
11    WHILE cardinality(open) > 0 LOOP
12        expr := open[1]; open := open[2:];
13
14        IF expr->>'entry' = 'num' THEN CONTINUE;
15        ELSIF expr->>'entry' = 'op' THEN
16            formulae := (SELECT array_agg(f) FROM jsonb_array_elements(expr->>'args') AS f);
17            open := open || formulae;
18        ELSIF expr->>'entry' = 'agg' THEN
19            args := (SELECT (array_agg(s.cell), array_agg(s.formula)) :: arguments
20                    FROM sheet AS s
21                    WHERE s.cell BETWEEN (expr->>'from') :: cell AND (expr->>'to') :: cell);
22            deps := args.cells || deps;
23            open := open || args.formulae;
24        ELSIF expr->>'entry' = 'cell' THEN
25            c := (expr->>'cell') :: cell;
26            deps := c || deps;
27            expr := (SELECT s.formula FROM sheet AS s WHERE s.cell = c);
28            open := open || expr;
29        END IF;
30    END LOOP;
31
32    -- intermediate cell contents found during evaluation
33    intermediates := array[] :: contents[];
34
35    -- 2 evaluate all relevant cells in dependency-order
36    FOREACH dep IN ARRAY deps LOOP
37        -- do not recompute known results
38        IF EXISTS(SELECT 1 FROM unnest(intermediates) AS i(c,v) WHERE i.c = dep) THEN CONTINUE; END IF;
39
40        e := (SELECT s.formula FROM sheet AS s WHERE s.cell = dep);
41        -- 2.1 transform expression tree into post-order
42        rpn := array[e]; exprs := array[] :: jsonb[];
43        WHILE cardinality(rpn) > 0 LOOP
44            root := rpn[1]; rpn := rpn[2:]; exprs := root || exprs;
45            IF root->>'entry' = 'num' THEN CONTINUE;
46            ELSIF root->>'entry' = 'op' THEN rpn := (SELECT array_agg(f)
47                    FROM jsonb_array_elements(root->>'args') AS f) || rpn;
48            ELSIF root->>'entry' = 'agg' THEN CONTINUE; ELSIF root->>'entry' = 'cell' THEN CONTINUE;
49            END IF;
50        END LOOP;
51
52        -- 2.2 evaluate post-order expression
53        stack := array[] :: float[];
54        FOREACH e in ARRAY exprs LOOP
55            IF e->>'entry' = 'num' THEN stack := (e->>'num') :: float || stack;
56            ELSIF e->>'entry' = 'op' THEN
57                IF e->>'op' = '+' THEN stack := (stack[1] + stack[2]) || stack[3:];
58                ELSIF e->>'op' = '-' THEN stack := (stack[1] - stack[2]) || stack[3:];
59                ELSIF e->>'op' = '*' THEN stack := (stack[1] * stack[2]) || stack[3:];
60                ELSIF e->>'op' = '/' THEN stack := (stack[1] / stack[2]) || stack[3:];
61            END IF;
62            ELSIF e->>'entry' = 'agg' THEN
63                stack := (SELECT CASE e->>'agg' WHEN 'sum' THEN SUM(i.v) WHEN 'avg' THEN AVG(i.v)
64                        WHEN 'max' THEN MAX(i.v) WHEN 'min' THEN MIN(i.v) END
65                        FROM unnest(intermediates) AS i(c,v)
66                        WHERE i.c BETWEEN (e->>'from') :: cell AND (e->>'to') :: cell) || stack;
67            ELSIF e->>'entry' = 'cell' THEN
68                stack := (SELECT i.v FROM unnest(intermediates) AS i(c,v) WHERE i.c = (e->>'cell') :: cell) || stack;
69            END IF;
70        END LOOP;
71
72        -- 3 save resulting cell value as intermediate result
73        intermediates := intermediates || (dep, stack[1]) :: contents[];
74    END LOOP;
75    -- 4 final cell value found in top of stack after formula evaluation
76    RETURN stack[1];
77 END;
78 $$ LANGUAGE PLPGSQL;

```

Function ship.

```

1 CREATE FUNCTION preferred_shipmode(custkey int) RETURNS TEXT AS
2 $$
3 DECLARE
4     ground int; air int; mail int;
5 BEGIN
6     -- collect shipping mode statistics
7     ground := (SELECT COUNT(*)
8               FROM lineitem AS l, orders AS o
9               WHERE l.l_orderkey = o.o_orderkey AND o.o_custkey = custkey AND l.l_shipmode IN ('RAIL', 'TRUCK'));
10    air := (SELECT COUNT(*)
11           FROM lineitem AS l, orders AS o
12           WHERE l.l_orderkey = o.o_orderkey AND o.o_custkey = custkey AND l.l_shipmode IN ('AIR', 'REG AIR'));
13    mail := (SELECT COUNT(*)
14            FROM lineitem AS l, orders AS o
15            WHERE l.l_orderkey = o.o_orderkey AND o.o_custkey = custkey AND l.l_shipmode = 'MAIL');
16    -- determine preferred shipping mode
17    IF ground >= air AND ground >= mail THEN RETURN 'ground';
18    ELSIF air >= ground AND air >= mail THEN RETURN 'air';
19    ELSIF mail >= ground AND mail >= air THEN RETURN 'mail';
20    END IF;
21    -- not reached
22    RETURN NULL;
23 END;
24 $$ LANGUAGE PLPGSQL;

```

Function sight.

```

1 CREATE FUNCTION sight(light point) RETURNS polygon AS
2 $$
3 DECLARE
4     points      point[];
5     p0          point;
6     p1          point;
7     p2          point;
8     phi         float := 0.001; -- ray angle offset ±( from p1, in radians)
9     target      point;
10    ins          point;
11    intersections point[];
12 BEGIN
13     -- ❶ edge points of all polygons
14     points := (SELECT array_agg(pt) FROM scene AS s, LATERAL unnest(points(s.poly)) AS pt);
15
16     intersections := array[] :: point[];
17
18     -- ❷ find intersection points of rays from light to all polygon edge points (+ jittering)
19     FOREACH p1 IN ARRAY points LOOP
20         p0 := point(light[0] + (p1[0] - light[0]) * cos(phi) - (p1[1] - light[1]) * sin(phi),
21                       light[1] + (p1[0] - light[0]) * sin(phi) + (p1[1] - light[1]) * cos(phi));
22         p2 := point(light[0] + (p1[0] - light[0]) * cos(-phi) - (p1[1] - light[1]) * sin(-phi),
23                       light[1] + (p1[0] - light[0]) * sin(-phi) + (p1[1] - light[1]) * cos(-phi));
24
25         FOREACH target in ARRAY array[p0,p1,p2] LOOP
26             ins := (SELECT ray(light, target) # lseg(seg0, seg1)
27                   FROM scene AS s, LATERAL points(s.poly) AS pts,
28                   LATERAL ROWS FROM (unnest(pts), unnest(pts[2:] || pts[1])) AS _(seg0,seg1)
29                   ORDER BY light <-> (ray(light, target) # lseg(seg0, seg1))
30                   LIMIT 1);
31
32             intersections := intersections || ins;
33         END LOOP;
34     END LOOP;
35
36     -- ❸ sort intersection points by angle
37     intersections := (SELECT array_agg(i ORDER BY degrees(atan2(light[0] - i[0], light[1] - i[1])))
38                     FROM unnest(intersections) AS i
39                     WHERE i IS NOT NULL);
40
41     RETURN polygon(intersections);
42 END;
43 $$
44 LANGUAGE PLPGSQL;

```

Function visible.

```

1 CREATE FUNCTION "visible?"(here point, there point, gridx int, gridy int, resolution int) RETURNS boolean AS
2 $$
3 DECLARE
4     step      point; -- direction of MAX scan
5     loc       point; -- current point of MAX scan
6     hhere     float; -- height at point here
7     hloc      float; -- height of current point loc during MAX scan
8     angle     float; -- angle between point here and current point of MAX scan
9     max_angle float; -- maximum angle measured during MAX scan
10 BEGIN
11     -- extent of landscape in x/y dimensions
12     gridx := gridx - 1;
13     gridy := gridy - 1;
14     -- height of point here (see https://en.wikipedia.org/wiki/Bézier\_surface)
15     hhere := (SELECT SUM((factorial(gridx) / (factorial(s.x) * ((factorial(gridx - s.x))))
16                 * u^s.x * (1 - u)^(gridx - s.x) *
17                 (factorial(gridy) / (factorial(s.y) * ((factorial(gridy - s.y))))
18                 * v^s.y * (1 - v)^(gridy - s.y) * h) AS h
19     FROM -- iterate over all points (s.x,s.y) of surface
20         (SELECT x, y
21          FROM generate_series(0, gridx) AS x, generate_series(0, gridy) AS y) AS s(x,y)
22          -- add control points (c.x,c.y) where there are defined
23          LEFT JOIN controlp AS c ON (c.x,c.y) = (s.x,s.y),
24          LATERAL (VALUES ( (here[0] / gridx) :: numeric
25                        , (here[1] / gridy :: numeric), COALESCE(c.z, 0))) AS _(u,v,h));
26
27     step      := (there - here) / resolution;
28     loc       := here;
29     -- maximum angle observed so far unknown
30     max_angle := NULL :: float;
31     -- perform a MAX scan along the line from here to there
32     FOR i IN 1..resolution LOOP
33         -- compute height at current location loc in scan
34         loc := loc + step;
35         hloc := (SELECT SUM((factorial(gridx) / (factorial(s.x) * ((factorial(gridx - s.x))))
36                 * u^s.x * (1 - u)^(gridx - s.x) *
37                 (factorial(gridy) / (factorial(s.y) * ((factorial(gridy - s.y))))
38                 * v^s.y * (1 - v)^(gridy - s.y) * h) AS h
39     FROM -- iterate over all points (s.x,s.y) of surface
40         (SELECT x, y
41          FROM generate_series(0, gridx) AS x, generate_series(0, gridy) AS y) AS s(x,y)
42          -- add control points (c.x,c.y) where there are defined
43          LEFT JOIN controlp AS c ON (c.x,c.y) = (s.x,s.y),
44          LATERAL (VALUES ( (loc[0] / gridx) :: numeric
45                        , (loc[1] / gridy) :: numeric, COALESCE(c.z, 0))) AS _(u,v,h));
46
47         -- viewing angle between here and current location of MAX scan
48         angle := degrees(atan((hloc - hhere) / (loc <-> here)));
49         -- save MAX angle observed during the scan
50         IF max_angle IS NULL OR angle > max_angle THEN max_angle := angle; END IF;
51     END LOOP;
52
53     -- point there is visible from here if its viewing angle is maximal
54     RETURN angle = max_angle;
55 END;
56 $$
57 LANGUAGE PLPGSQL;

```


Function vm.

```

1 CREATE FUNCTION vm(regs numeric[]) RETURNS numeric AS
2 $$
3 DECLARE
4     ip int := 0;
5     ins instruction;
6 BEGIN
7     LOOP
8         ins := (SELECT p FROM program AS p WHERE p.loc = ip);
9         ip := ip + 1;
10
11        IF ins.opc = 'lod' THEN regs := regs[:ins.reg1-1] || ins.reg2 || regs[ins.reg1+1:];
12        ELSIF ins.opc = 'mov' THEN regs := regs[:ins.reg1-1] || regs[ins.reg2] || regs[ins.reg1+1:];
13        ELSIF ins.opc = 'jeq' THEN IF regs[ins.reg1] = regs[ins.reg2] THEN ip := ins.reg3 :: int4; END IF;
14        ELSIF ins.opc = 'jmp' THEN ip := ins.reg1 :: int4;
15        ELSIF ins.opc = 'add' THEN regs := regs[:ins.reg1-1] || regs[ins.reg2] + regs[ins.reg3] || regs[ins.reg1+1:];
16        ELSIF ins.opc = 'sub' THEN regs := regs[:ins.reg1-1] || regs[ins.reg2] - regs[ins.reg3] || regs[ins.reg1+1:];
17        ELSIF ins.opc = 'mul' THEN regs := regs[:ins.reg1-1] || regs[ins.reg2] * regs[ins.reg3] || regs[ins.reg1+1:];
18        ELSIF ins.opc = 'div' THEN regs := regs[:ins.reg1-1] || regs[ins.reg2] / regs[ins.reg3] || regs[ins.reg1+1:];
19        ELSIF ins.opc = 'mod' THEN regs := regs[:ins.reg1-1] || regs[ins.reg2] % regs[ins.reg3] || regs[ins.reg1+1:];
20        ELSIF ins.opc = 'hlt' THEN RETURN regs[ins.reg1];
21        END IF;
22
23    END LOOP;
24 END;
25 $$
26 LANGUAGE PLPGSQL;

```

Function ray.

```

1 CREATE FUNCTION ray(w int, h int) RETURNS int[] AS
2 $$
3 DECLARE
4     cam          vec3 := (0.0, 0.0, -4.5); cam_lookat vec3 := (0.0, 0.0, 0.0); cam_up vec3 := (0.0, 1.0, 0.0);
5     light        vec3; light_r real; fov real := 50.0; shadows boolean := true; max_rec_depth int := 10;
6     aspect_ratio real := w / h;
7     epsilon      real := 0.00001; ntriangles int; nspheres int; nprimitives int; cd vec3; tlen real;
8     rotx1        real; rotx2 real; rotx3 real; roty1 real; roty2 real; roty3 real; rotz1 real; rotz2 real; rotz3 real;
9     degx         real; degy real; t vec3; rd vec3; ro vec3; do_ray boolean; shadow_done boolean; c rgb; intersection boolean;
10    mindist real; material material; mat material; ho rgb; col rgb; no vec3; triangle triangles; sphere spheres; prim_hit boolean;
11    v1 vec3; v2 vec3; v3 vec3; e1 vec3; e2 vec3; P vec3; T1 vec3; Q vec3; det real; u real; v real; tdist real; tdot real; sp vec3;
12    spr real; L vec3; tca real; d2 real; thc real; r int[] := array[] :: int[];
13 BEGIN
14    ntriangles := (SELECT COUNT(*) FROM triangles); nspheres := (SELECT COUNT(*) FROM spheres); nprimitives := ntriangles + nspheres;
15    -- find light source among spheres
16    sphere := (SELECT sph FROM spheres AS sph WHERE sph.mat = '1'); light := sphere.center; light_r := sphere.radius;
17    fov := fov * (pi() / 180.0); cd := (cam_lookat.x - cam.x, cam_lookat.y - cam.y, cam_lookat.z - cam.z);
18    tlen := sqrt(cd.x2 + cd.y2 + cd.z2); rotx1 := cd.x / tlen; rotx2 := cd.y / tlen; rotx3 := cd.z / tlen;
19    rotx1 := cam_up.y * rotx2 - cam_up.z * rotx3; rotx2 := cam_up.z * rotx1 - cam_up.x * rotx3;
20    rotx3 := cam_up.x * rotx2 - cam_up.y * rotx1; tlen := sqrt(rotx12 + rotx22 + rotx32); rotx1 := rotx1 / tlen;
21    rotx2 := rotx2 / tlen; rotx3 := rotx3 / tlen; roty1 := rotx2 * rotx3 - rotx3 * rotx2; roty2 := rotx3 * rotx1 - rotx1 * rotx3;
22    roty3 := rotx1 * rotx2 - rotx2 * rotx1;
23    FOR pxy IN 0 .. h-1 LOOP
24        FOR pxx IN 0 .. w-1 LOOP
25            degx := (((pxx + 0.5) / w) - 0.5) * fov * aspect_ratio; degy := (((pxy + 0.5) / h) - 0.5) * fov;
26            t := (sin(degx), sin(degy), 1.0);
27            rd := (t.x*rotx1 + t.y*roty1 + t.z*rotx1, t.x*rotx2 + t.y*roty2 + t.z*rotx2, t.x*rotx3 + t.y*roty3 + t.z*rotx3);
28            ro := cam; do_ray := true; shadow_done := false; c := (0.0, 0.0, 0.0);
29            FOR rec IN 1 .. (1 + max_rec_depth + shadows :: int) LOOP
30                IF do_ray THEN
31                    do_ray := false; tlen := sqrt(rd.x2 + rd.y2 + rd.z2); rd := (rd.x / tlen, rd.y / tlen, rd.z / tlen);
32                    intersection := false; mindist := 999999; material := 'n'; no := (0.0, 0.0, 0.0);
33                    FOR i IN 1 .. nprimitives LOOP
34                        prim_hit := false;
35                        IF i <= ntriangles THEN
36                            triangle := (SELECT tri FROM triangles AS tri WHERE tri.id = i); v1 := triangle.p1; v2 := triangle.p2; v3 := triangle.p3;
37                            mat := triangle.mat; col := triangle.color;
38                            e1 := (v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); e2 := (v3.x - v1.x, v3.y - v1.y, v3.z - v1.z);
39                            P := (rd.y*e2.z - rd.z*e2.y, rd.z*e2.x - rd.x*e2.z, rd.x*e2.y - rd.y*e2.x); det := e1.x*P.x + e1.y*P.y + e1.z*P.z;
40                            IF abs(det) > epsilon THEN
41                                det := 1.0 / det; T1 := (ro.x - v1.x, ro.y - v1.y, ro.z - v1.z); u := (T1.x*P.x + T1.y*P.y + T1.z*P.z) * det;
42                                IF u BETWEEN 0.0 AND 1.0 THEN
43                                    Q := (T1.y*e1.z - T1.z*e1.y, T1.z*e1.x - T1.x*e1.z, T1.x*e1.y - T1.y*e1.x); v := (rd.x*Q.x + rd.y*Q.y + rd.z*Q.z) * det;
44                                    IF v >= 0.0 AND u + v <= 1.0 THEN
45                                        tdist := (e2.x*Q.x + e2.y*Q.y + e2.z*Q.z) * det;
46                                        IF tdist > epsilon AND tdist < mindist THEN
47                                            prim_hit := true; intersection := true; mindist := tdist;
48                                            no := (e2.y*e1.z - e2.z*e1.y, e2.z*e1.x - e2.x*e1.z, e2.x*e1.y - e2.y*e1.x);
49                                            tlen := sqrt(no.x2 + no.y2 + no.z2); no := (no.x / tlen, no.y / tlen, no.z / tlen);
50                                            tdot := no.x*rd.x + no.y*rd.y + no.z*rd.z;
51                                            IF tdot > 0.0 THEN no := (-no.x, -no.y, -no.z); END IF; -- tdot > 0.0
52                                            END IF; -- tdist > epsilon AND tdist < mindist
53                                            END IF; -- v >= 0.0 AND u + v <= 1.0
54                                            END IF; -- u BETWEEN 0.0 AND 1.0
55                                            END IF; -- abs(det) > epsilon
56                                        ELSE
57                                            sphere := (SELECT sph FROM spheres AS sph WHERE sph.id = i - ntriangles);
58                                            sp := sphere.center; spr := sphere.radius; mat := sphere.mat; col := sphere.color;
59                                            L := (sp.x - ro.x, sp.y - ro.y, sp.z - ro.z); tca := L.x*rd.x + L.y*rd.y + L.z*rd.z; d2 := L.x2 + L.y2 + L.z2 - tca2;
60                                            IF d2 <= spr2 THEN
61                                                thc := sqrt(spr2 - d2); tdist := 0.0;
62                                                IF tca - thc > 0.0 THEN tdist := tca - thc; END IF;
63                                                IF tca + thc > 0.0 THEN tdist := LEAST(tca + thc, tdist); END IF;
64                                                IF tdist > 0.0 AND tdist < mindist THEN
65                                                    prim_hit := true; intersection := true; mindist := tdist;
66                                                    no := (ro.x*tdist*rd.x - sp.x, ro.y*tdist*rd.y - sp.y, ro.z*tdist*rd.z - sp.z); tlen := sqrt(no.x2 + no.y2 + no.z2);
67                                                    no := (no.x / tlen, no.y / tlen, no.z / tlen);
68                                                    END IF; -- tdist > 0.0 AND tdist < mindist
69                                                    END IF; -- d2 <= spr2
70                                                ELSE
71                                                    IF prim_hit THEN material := mat; -- if no primitive was hit by ray, material remains 'n'
72                                                    IF material = 'm' THEN ho := col; END IF;
73                                                    END IF; -- prim_hit
74                                                END LOOP; -- i
75                                            IF shadow_done THEN IF material <> '1' THEN c := (0.0, 0.0, 0.0); END IF; -- material <> '1'
76                                            ELSE
77                                                IF material = '1' THEN c := (1.0, 1.0, 1.0); END IF; -- material = '1'
78                                                IF material = 'm' THEN
79                                                    li := (light.x - (ro.x + rd.x * mindist), light.y - (ro.y + rd.y * mindist), light.z - (ro.z + rd.z * mindist));
80                                                    tlen := sqrt(li.x2 + li.y2 + li.z2); li := (li.x / tlen, li.y / tlen, li.z / tlen);
81                                                    tdot := GREATEST(0.0, li.x*no.x + li.y*no.y + li.z*no.z); c := (ho.r * tdot, ho.g * tdot, ho.b * tdot);
82                                                    IF shadows AND NOT shadow_done THEN
83                                                        ro := (ro.x + rd.x * mindist * no.x * epsilon, ro.y + rd.y * mindist * no.y * epsilon, ro.z + rd.z * mindist * no.z * epsilon);
84                                                        shadow_done := true; rd := (light.x - ro.x, light.y - ro.y, light.z - ro.z); do_ray := true;
85                                                    END IF; -- shadows AND NOT shadow_done
86                                                    END IF; -- material = 'm'
87                                                IF material = 'r' THEN
88                                                    tdot := rd.x*no.x + rd.y*no.y + rd.z*no.z;
89                                                    ro := (ro.x + rd.x * mindist * no.x * epsilon, ro.y + rd.y * mindist * no.y * epsilon, ro.z + rd.z * mindist * no.z * epsilon);
90                                                    rd := (rd.x - 2.0 * no.x * tdot, rd.y - 2.0 * no.y * tdot, rd.z - 2.0 * no.z * tdot);
91                                                    do_ray := true;
92                                                    END IF; -- material = 'r'
93                                                END IF; -- shadow_done
94                                            END IF; -- do_ray
95                                        END LOOP; -- rec
96                                        IF intersection THEN r := r || array[(c.b * 255) :: int, (c.g * 255) :: int, (c.r * 255) :: int];
97                                        ELSE r := r || array[0, 0, 0];
98                                        END IF; -- intersection
99                                    END LOOP; -- pxx
100                                END LOOP; -- pxy
101                                RETURN r;
102                                END;
103                                $$ LANGUAGE PLPGSQL;

```

Recursive UDF Definitions

B.

In the following we list the recursive UDF definitions of the UDFs in Table 10.1. The definitions are given in SQL syntax. We use the `CREATE FUNCTION` statement to define the UDFs.

All UDFs are available with the necessary setup code here: <https://github.com/FP-on-Top-of-SQL-Engines/Code>

Function comps.

```
1 CREATE FUNCTION connected(node int, target int) RETURNS boolean AS $$
2 SELECT CASE
3     -- Components are connected.
4     WHEN node = target THEN TRUE
5     -- Reached a leaf without having found the target we are looking for
6     WHEN NOT EXISTS (SELECT n.id FROM nodes AS n WHERE n.id = node)
7     THEN FALSE
8     -- We found two children and thus, continue
9     -- to recurse with both child 'l' and 'r' as arguments.
10    WHEN (SELECT COUNT(*) FROM nodes AS n WHERE n.id = node) = 2
11    THEN connected((SELECT n.next FROM nodes AS n
12                    WHERE (n.id, n.child) = (node, 'l')), target) OR
13    connected((SELECT n.next FROM nodes AS n
14              WHERE (n.id, n.child) = (node, 'r')), target)
15    ELSE -- Only one child was found.
16    connected((SELECT n.next FROM nodes AS n WHERE n.id = node), target)
17 END;
18 $$ LANGUAGE SQL STABLE STRICT;
```

Function dtw.

```
1 CREATE FUNCTION dtw(i int, j int) RETURNS double precision AS $$
2 SELECT CASE
3     WHEN i = 0 AND j = 0 THEN 0
4     WHEN (i <> 0 AND j = 0) OR (i = 0 AND j <> 0)
5     THEN 'Infinity' :: double precision
6     WHEN i <> 0 AND j <> 0
7     THEN (SELECT ABS(X.x-Y.y) + LEAST(dtw(i-1, j-1), dtw(i-1, j), dtw(i, j-1))
8         FROM X, Y
9         WHERE (X.t, Y.t) = (i, j))
10    END;
11 $$ LANGUAGE SQL STABLE STRICT;
```

Function eval.

```
1 CREATE FUNCTION eval(e expression) RETURNS numeric AS $$
2 SELECT CASE e.op
3     WHEN 'l' THEN e.lit
4     WHEN '+'
5     THEN eval((SELECT e1 FROM expression AS e1 WHERE e1.node = e.arg1))
6     +
7     eval((SELECT e2 FROM expression AS e2 WHERE e2.node = e.arg2))
8     WHEN '*'
9     THEN eval((SELECT e1 FROM expression AS e1 WHERE e1.node = e.arg1))
10    *
11    eval((SELECT e2 FROM expression AS e2 WHERE e2.node = e.arg2))
12 END;
13 $$ LANGUAGE SQL;
```

Function fsm.

```

1 CREATE FUNCTION parse(state int, input text) RETURNS boolean AS $$
2   SELECT CASE WHEN length(input) = 0
3     THEN (SELECT DISTINCT edge.final
4           FROM fsm AS edge
5           WHERE state = edge.source)
6     ELSE COALESCE(parse((
7       SELECT edge.target
8       FROM fsm AS edge
9       WHERE state = edge.source
10      AND strpos(edge.labels, left(input, 1)) > 0
11      ), right(input, -1)), false)
12   END;
13 $$ LANGUAGE SQL;

```

Function lcs.

```

1 CREATE FUNCTION lcs(l text, r text) RETURNS int AS $$
2   SELECT CASE
3     WHEN l = '' OR r = '' THEN 0
4     WHEN left(l,1) = left(r,1) THEN 1 + lcs(right(l,-1), right(r,-1))
5     ELSE GREATEST(lcs(right(l,-1), r), lcs(l, right(r,-1)))
6   END;
7 $$ LANGUAGE SQL STABLE STRICT;

```

Function mbrot.

```

1 CREATE FUNCTION m(iter int, cx float, cy float,
2   x float, y float) RETURNS int AS $$
3   SELECT CASE WHEN NOT (x2 + y2 < 4.0 AND iter < 28)
4     THEN iter
5     ELSE m(iter + 1,
6           cx,
7           cy,
8           x2 - y2 + cx,
9           2.0 * x * y + cy)
10  END;
11 $$
12 LANGUAGE SQL IMMUTABLE STRICT;

```

Function paths.

```

1 CREATE FUNCTION file_path(dir text, file_path text) RETURNS text AS $$
2   SELECT CASE
3     WHEN (SELECT d.PARENT_DIR_ID
4           FROM DIRS AS d
5           WHERE d.DIR_NAME = dir) IS NULL THEN '/' || dir || file_path
6     ELSE file_path((
7       SELECT d2.DIR_NAME
8       FROM DIRS AS d, DIRS AS d2
9       WHERE d.DIR_NAME = dir
10      AND d.PARENT_DIR_ID = d2.DIR_ID),
11      '/' || dir || file_path)
12   END;
13 $$ LANGUAGE SQL STABLE STRICT;

```

Function vm.

```

1 CREATE FUNCTION run(ins instruction, regs int[]) RETURNS int AS $$
2   SELECT CASE ins.opc
3     WHEN 'lod'
4       THEN run((SELECT p FROM program AS p WHERE p.loc = ins.loc+1),
5                regs[:ins.reg1-1] || ins.reg2 || regs[ins.reg1+1:])
6
7     WHEN 'mov'
8       THEN run((SELECT p FROM program AS p WHERE p.loc = ins.loc+1),
9                regs[:ins.reg1-1] || regs[ins.reg2] || regs[ins.reg1+1:])
10
11    WHEN 'jeq'
12      THEN run((SELECT p FROM program AS p
13                WHERE p.loc = CASE WHEN regs[ins.reg1] = regs[ins.reg2]
14                                THEN ins.reg3
15                                ELSE ins.loc + 1
16                                END), regs)
17
18    WHEN 'jmp'
19      THEN run((SELECT p FROM program AS p WHERE p.loc = ins.reg1), regs)
20
21    WHEN 'add'
22      THEN run((SELECT p FROM program AS p WHERE p.loc = ins.loc+1),
23                regs[:ins.reg1-1] ||
24                regs[ins.reg2] + regs[ins.reg3] ||
25                regs[ins.reg1+1:])
26
27    WHEN 'mul'
28      THEN run((SELECT p FROM program AS p WHERE p.loc = ins.loc+1),
29                regs[:ins.reg1-1] ||
30                regs[ins.reg2] * regs[ins.reg3] ||
31                regs[ins.reg1+1:])
32
33    WHEN 'div'
34      THEN run((SELECT p FROM program AS p WHERE p.loc = ins.loc+1),
35                regs[:ins.reg1-1] ||
36                regs[ins.reg2] / regs[ins.reg3] ||
37                regs[ins.reg1+1:])
38
39    WHEN 'mod'
40      THEN run((SELECT p FROM program AS p WHERE p.loc = ins.loc+1),
41                regs[:ins.reg1-1] ||
42                regs[ins.reg2] % regs[ins.reg3] ||
43                regs[ins.reg1+1:])
44
45    WHEN 'hlt' THEN regs[ins.reg1]
46  END
47 $$
48 LANGUAGE SQL STABLE STRICT;

```


Bibliography

Here are the references in citation order.

- [1] Donald D. Chamberlin and Raymond F. Boyce. ‘SEQUEL: A Structured English Query Language’. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’74. Ann Arbor, Michigan: Association for Computing Machinery, 1974, pp. 249–264. doi: [10.1145/800296.811515](https://doi.org/10.1145/800296.811515) (cited on page 1).
- [2] Steven Feuerstein and Bill Pribyl. *Oracle PL/SQL Programming*. ”O’Reilly Media, Inc.”, 2014 (cited on page 1).
- [3] George P. Copeland and David Maier. ‘Making smalltalk a database system’. In: *SIGMOD ’84*. 1984 (cited on page 2).
- [4] *Oracle 19c PL/SQL Documentation*. <http://docs.oracle.com/en/database/oracle/oracle-database/19/lmpls> (cited on page 2).
- [5] Diego Novillo. ‘Tree SSA a new optimization infrastructure for GCC’. In: *Proceedings of the 2003 gCC developers’ summit*. Citeseer. 2003, pp. 181–193 (cited on pages 5, 30).
- [6] Chris Arthur Lattner. ‘LLVM: An infrastructure for multi-stage optimization’. PhD thesis. University of Illinois at Urbana-Champaign, 2002 (cited on page 5).
- [7] Luke Maurer et al. ‘Administrative normal form, continued’. In: (2017) (cited on page 5).
- [8] Luke Maurer et al. ‘Compiling without Continuations’. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 482–494. doi: [10.1145/3062341.3062380](https://doi.org/10.1145/3062341.3062380) (cited on page 5).
- [9] Manuel MT Chakravarty, Gabriele Keller, and Patryk Zadarnowski. ‘A functional perspective on SSA optimisation algorithms’. In: *Electronic Notes in Theoretical Computer Science* 82.2 (2004), pp. 347–361 (cited on pages 5, 32, 47, 64, 80).
- [10] Steven E Ganz, Daniel P Friedman, and Mitchell Wand. ‘Trampolined style’. In: *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*. 1999, pp. 18–27 (cited on pages 5, 34, 35, 75).
- [11] David Tarditi, Peter Lee, and Anurag Acharya. ‘No Assembly Required: Compiling Standard ML to C’. In: *ACM Lett. Program. Lang. Syst.* 1.2 (1992), pp. 161–177. doi: [10.1145/151333.151343](https://doi.org/10.1145/151333.151343) (cited on page 5).
- [12] Michel Schinz and Martin Odersky. ‘Tail call elimination on the Java Virtual Machine’. In: *Electronic Notes in Theoretical Computer Science* 59.1 (2001), pp. 158–171 (cited on page 5).
- [13] Christian Duta, Denis Hirn, and Torsten Grust. ‘Compiling PL/SQL Away’. In: *Proc. CIDR*. 2020 (cited on pages 6, 14, 42, 60, 63).
- [14] Denis Hirn and Torsten Grust. ‘PL/SQL Without the PL’. In: *Proc. SIGMOD*. 2020 (cited on pages 6, 14, 59, 60, 63).
- [15] Denis Hirn and Torsten Grust. ‘One WITH RECURSIVE is Worth Many GOTOs’. In: *Proc. SIGMOD*. 2021 (cited on pages 6, 14, 42, 56, 60, 63, 73).
- [16] Denis Hirn and Torsten Grust. ‘A Fix for the Fixation on Fixpoints’. In: *Proc. CIDR*. 2023 (cited on pages 6, 64).
- [17] Denis Hirn. ‘Data is Data and Control Should be Data, Too’. In: *Proc. VLDB*. 2023 (cited on page 6).

- [18] Olivier Danvy. ‘Back to direct style’. In: *Science of Computer Programming* 22.3 (1994), pp. 183–195 (cited on page 7).
- [19] John C. Reynolds. ‘Definitional Interpreters for Higher-Order Programming Languages’. In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM ’72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740. doi: [10.1145/800194.805852](https://doi.org/10.1145/800194.805852) (cited on pages 7, 73, 75, 82).
- [20] Olivier Danvy and Lasse R. Nielsen. ‘Defunctionalization at Work’. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’01. Florence, Italy: Association for Computing Machinery, 2001, pp. 162–174. doi: [10.1145/773184.773202](https://doi.org/10.1145/773184.773202) (cited on page 7).
- [21] Tobias Burghardt, Denis Hirn, and Torsten Grust. ‘Functional Programming on Top of SQL Engines’. In: *Practical Aspects of Declarative Languages: 24th International Symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings*. Philadelphia, PA, USA: Springer-Verlag, 2022, pp. 59–78. doi: [10.1007/978-3-030-94479-7_5](https://doi.org/10.1007/978-3-030-94479-7_5) (cited on page 8).
- [22] L.A. Rowe and M. Stonebraker. ‘The POSTGRES Data Model’. In: *Proc. VLDB*. Brighton, UK, Sept. 1987 (cited on pages 13, 14, 75, 109).
- [23] Donald Bales. *Beginning Oracle PL/SQL*. Apress, 2015 (cited on page 13).
- [24] S. Gupta, S. Purandare, and K. Ramachandra. ‘Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates’. In: *Proc. SIGMOD*. 2020 (cited on pages 13, 14, 18, 66).
- [25] Shudi Shao et al. ‘Database-Access Performance Antipatterns in Database-Backed Web Applications’. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 58–69. doi: [10.1109/ICSME46990.2020.00016](https://doi.org/10.1109/ICSME46990.2020.00016) (cited on page 13).
- [26] Christopher Olston et al. ‘Pig Latin: A Not-so-Foreign Language for Data Processing’. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 1099–1110. doi: [10.1145/1376616.1376726](https://doi.org/10.1145/1376616.1376726) (cited on page 14).
- [27] K. Ramachandra et al. ‘Froid: Optimization of Imperative Programs in a Relational Database’. In: *Proc. VLDB* 11.4 (2018) (cited on pages 14, 18, 36, 53, 63, 65, 73, 109).
- [28] J.H. Harris. *A (Not So) Brief But (Very) Accurate History of PL/SQL*. <http://oracle-internals.com/blog/2020/04/29/a-not-so-brief-but-very-accurate-history-of-pl-sql/>. Apr. 2020 (cited on page 14).
- [29] Varun Simhadri et al. ‘Decorrelation of user defined function invocations in queries’. In: *2014 IEEE 30th International Conference on Data Engineering*. 2014, pp. 532–543. doi: [10.1109/ICDE.2014.6816679](https://doi.org/10.1109/ICDE.2014.6816679) (cited on page 14).
- [30] Timo Kersten et al. ‘Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask’. In: *Proc. VLDB Endow*. 11.13 (2018), pp. 2209–2222. doi: [10.14778/3275366.3275370](https://doi.org/10.14778/3275366.3275370) (cited on page 15).
- [31] Jeff Moden. *Calculating Work Days*. <https://web.archive.org/web/20071023061855/http://www.sqlservercentral.com/articles/Advanced+Querying/calculatingworkdays/1660/>. Jan. 2005 (cited on page 16).
- [32] Grant Fritchey and Grant Fritchey. ‘Row-by-Row Processing’. In: *SQL Server Query Performance Tuning* (2014), pp. 459–481 (cited on page 16).
- [33] *POSTGRESQL 15 Documentation*. <http://www.postgresql.org/docs/15/> (cited on pages 16, 18, 24, 27, 51, 75, 89).
- [34] K. Ramachandra and K. Park. ‘BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid’. In: *Proc. VLDB* 12.12 (2019) (cited on pages 18, 65).
- [35] *Microsoft SQL Server 2022 Documentation*. <http://docs.microsoft.com/en-us/sql> (cited on pages 18, 75).

- [36] *POSTGRES SQL 15 PL/PgSQL Documentation*. <http://www.postgresql.org/docs/15/plpgsql.html> (cited on pages 19, 26, 39).
- [37] A. Eisenberg and J. Melton. ‘SQL:1999, Formerly Known as SQL3’. In: *ACM SIGMOD Record* 28.1 (Mar. 1999) (cited on pages 23, 75).
- [38] *SQL:1999 Standard. Database Languages—SQL—Part 2: Foundation*. ISO/IEC 9075-2:1999 (cited on pages 23, 50, 75).
- [39] Galina Shalygina and Boris Novikov. ‘Implementing Common Table Expressions for MariaDB’. In: *Proceedings of the 2nd Conference on Software Engineering and Information Management (SEIM-2017), St. Petersburg, Russia*. Vol. 21. 2017 (cited on page 25).
- [40] Alfred Tarski. ‘A lattice-theoretical fixpoint theorem and its applications.’ In: (1955) (cited on page 25).
- [41] Francois Bancilhon and Raghu Ramakrishnan. ‘An Amateur’s Introduction to Recursive Query Processing Strategies’. In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’86. Washington, D.C., USA: Association for Computing Machinery, 1986, pp. 16–52. doi: [10.1145/16894.16859](https://doi.org/10.1145/16894.16859) (cited on page 25).
- [42] S.J. Finkelstein et al. *Expressive Recursive Queries in SQL*. Joint Technical Committee ISO/IEC JTC 1/SC 21 WG 3, Document X3H2-96-075r1. 1996 (cited on pages 25, 75).
- [43] Francois Bancilhon. ‘Naive Evaluation of Recursively Defined Relations’. In: *On Knowledge Base Management Systems: Integrating Artificial Intelligence and d Atabase Technologies*. Berlin, Heidelberg: Springer-Verlag, 1986, pp. 165–178 (cited on page 25).
- [44] Jim Gray. ‘The Transaction Concept: Virtues and Limitations (Invited Paper)’. In: *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB ’81. Cannes, France: VLDB Endowment, 1981, pp. 144–154 (cited on page 27).
- [45] Theo Haerder and Andreas Reuter. ‘Principles of Transaction-Oriented Database Recovery’. In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. doi: [10.1145/289.291](https://doi.org/10.1145/289.291) (cited on page 27).
- [46] Frances E Allen. ‘Control flow analysis’. In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19 (cited on page 29).
- [47] James Stanier and Des Watson. ‘Intermediate Representations in Imperative Compilers: A Survey’. In: *ACM Comput. Surv.* 45.3 (2013). doi: [10.1145/2480741.2480743](https://doi.org/10.1145/2480741.2480743) (cited on page 29).
- [48] Simon Peyton Jones and Simon Marlow. ‘Secrets of the Glasgow Haskell Compiler inliner’. In: *Journal of Functional Programming* 12 (July 2002), pp. 393–434 (cited on page 29).
- [49] P. P. Chang and W.-W. Hwu. ‘Inline Function Expansion for Compiling C Programs’. In: *Proc. PLDI*. Portland, OR, USA, June 1989 (cited on page 29).
- [50] J. Ferrante, Karl J. Ottenstein, and Joe D. Warren. ‘The Program Dependence Graph and Its Use in Optimization’. In: *ACM TOPLAS* 9.3 (July 1987) (cited on page 29).
- [51] O. Waddell and R.K. Dybig. ‘Fast and Effective Procedure Inlining’. In: *Proc. Int’l Symposium on Static Analysis*. Paris, France, Sept. 1997 (cited on page 29).
- [52] R. Cytron et al. ‘Efficiently Computing Static Single Assignment Form and the Control Dependence Graph’. In: *ACM TOPLAS* 13.4 (1991) (cited on pages 30, 45).
- [53] Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based Compiler Design*. Singapore: Springer Nature, 2022 (cited on pages 30, 78).
- [54] M. Braun et al. ‘Simple and Efficient Construction of Static Single Assignment Form’. In: *Proc. Int’l Conference on Compiler Construction*. Rome, Italy, Mar. 2013 (cited on pages 30, 45).
- [55] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. ‘Automatic Construction of Sparse Data Flow Evaluation Graphs’. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’91. Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 55–66. doi: [10.1145/99583.99594](https://doi.org/10.1145/99583.99594) (cited on page 30).

- [56] Preston Briggs et al. ‘Practical Improvements to the Construction and Destruction of Static Single Assignment Form’. In: *Softw. Pract. Exper.* 28.8 (1998), pp. 859–881 (cited on page 30).
- [57] A.W. Appel. ‘SSA is Functional Programming’. In: *ACM SIGPLAN Notices* 33.4 (Apr. 1998) (cited on pages 30, 32, 45, 64, 75).
- [58] Jeffrey C. Lagarias. ‘The $3x + 1$ Problem and its Generalizations’. In: *The American Mathematical Monthly* 92.1 (1985), pp. 3–23. doi: [10.1080/00029890.1985.11971528](https://doi.org/10.1080/00029890.1985.11971528) (cited on page 31).
- [59] Philip Wadler. ‘Monads for functional programming’. In: *Program Design Calculi*. Ed. by Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 233–264 (cited on page 31).
- [60] Richard A. Kelsey. ‘A Correspondence between Continuation Passing Style and Static Single Assignment Form’. In: *SIGPLAN Not.* 30.3 (1995), pp. 13–22. doi: [10.1145/202530.202532](https://doi.org/10.1145/202530.202532) (cited on page 32).
- [61] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997 (cited on page 32).
- [62] Cormac Flanagan et al. ‘The Essence of Compiling with Continuations’. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI ’93. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1993, pp. 237–247. doi: [10.1145/155090.155113](https://doi.org/10.1145/155090.155113) (cited on pages 32, 33).
- [63] Suresh Jagannathan and Andrew Wright. ‘Flow-directed inlining’. In: *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation - PLDI ’96* (1996). doi: [10.1145/231379.231417](https://doi.org/10.1145/231379.231417) (cited on page 33).
- [64] Kim Solin. ‘Normal forms in total correctness for while programs and action systems’. In: *The Journal of Logic and Algebraic Programming* 80.6 (2011), pp. 362–375 (cited on page 34).
- [65] C. Galindo-Legaria and M. Joshi. ‘Orthogonal Optimization of Subqueries and Aggregation’. In: *Proc. SIGMOD*. 2001 (cited on pages 37, 50).
- [66] C. Maple. ‘Geometric design and space planning using the marching squares and marching cube algorithms’. In: *Proc. GMAG 2003*. 2003. doi: [10.1109/GMAG.2003.1219671](https://doi.org/10.1109/GMAG.2003.1219671) (cited on pages 39, 95).
- [67] *Oracle Database PL/SQL Language Reference 21c*. <https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/database-pl-sql-language-reference.pdf> (cited on pages 51, 75).
- [68] Mark Raasveldt and Hannes Mühleisen. ‘DuckDB: an Embeddable Analytical Database’. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1981–1984 (cited on pages 51, 110, 111).
- [69] T.J. McCabe. ‘A Complexity Measure’. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320 (cited on page 53).
- [70] R. Guravannavar and S. Sudarshan. ‘Rewriting Procedures for Batched Bindings’. In: *Proc. VLDB* 1.1 (2008) (cited on page 54).
- [71] V. Simhadri et al. ‘Decorrelation of User Defined Functions in Queries’. In: *Proc. ICDE*. Chicago, IL, USA, Mar. 2014 (cited on pages 54, 66).
- [72] Holtsetio. *MySQL Raytracer*. <https://demozoo.org/productions/268459/>. Oct. 2019 (cited on page 54).
- [73] J. Barnes and P. Hut. ‘A Hierarchical $O(N \log N)$ Force-Calculation Algorithm’. In: *Nature* 324.4 (1986) (cited on page 55).
- [74] P. Griffiths Selinger et al. ‘Access Path Selection in a Relational Database Management System’. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’79. Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34. doi: [10.1145/582095.582099](https://doi.org/10.1145/582095.582099) (cited on page 56).

- [75] Jayant R. Haritsa. ‘The Picasso Database Query Optimizer Visualizer’. In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 1517–1520. doi: [10.14778/1920841.1921027](https://doi.org/10.14778/1920841.1921027) (cited on page 56).
- [76] M. Abhirama et al. ‘On the Stability of Plan Costs and the Costs of Plan Stability’. In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 1137–1148. doi: [10.14778/1920841.1920983](https://doi.org/10.14778/1920841.1920983) (cited on page 56).
- [77] T.K. Sellis. ‘Multiple-Query Optimization’. In: *ACM TODS* 13.1 (Mar. 1998) (cited on page 58).
- [78] G. Graefe. ‘Volcano—An Extensible and Parallel Query Evaluation System’. In: *IEEE TKDE* 6.1 (Feb. 1994) (cited on pages 60, 74).
- [79] M. Boehm, A. Kumar, and J. Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool, 2019 (cited on page 63).
- [80] D. Jankov et al. ‘Declarative Recursive Computation on an RDBMS (or Why You Should Use a Database for Distributed Machine Learning)’. In: *Proc. VLDB* 12.7 (2019) (cited on page 63).
- [81] X. Feng et al. ‘Towards a Unified Architecture for in-RDBMS Analytics’. In: *Proc. SIGMOD*. Scottsdale, AZ, USA, May 2012 (cited on page 63).
- [82] Mark Blacher et al. ‘Machine learning, linear algebra, and more: Is SQL all you need’. In: *CIDR*. www.cidrdb.org (2022), pp. 1–6 (cited on pages 63, 69).
- [83] K.V. Emani et al. ‘Extracting Equivalent SQL from Imperative Code in Database Applications’. In: *Proc. SIGMOD*. San Francisco, CA, USA, June 2016 (cited on page 63).
- [84] Torsten Grust, Jan Rittinger, and Tom Schreiber. ‘Avalanche-Safe LINQ Compilation’. In: *Proc. VLDB* 3.1 (Sept. 2010) (cited on page 63).
- [85] C. Lattner et al. ‘MLIR: A Compiler Infrastructure for the End of Moore’s Law’. In: *CoRR* (*arXiv*) abs/2002.11054 (2020) (cited on page 64).
- [86] Jin Wang et al. ‘RASQL: A Powerful Language and Its System for Big Data Applications’. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2673–2676. doi: [10.1145/3318464.3384677](https://doi.org/10.1145/3318464.3384677) (cited on page 64).
- [87] Linnea Passing et al. ‘SQL-and Operator-centric Data Analytics in Relational Main-Memory Databases.’ In: *EDBT*. 2017, pp. 84–95 (cited on page 64).
- [88] Thomas Neumann and Michael J Freitag. ‘Umbra: A Disk-Based System with In-Memory Performance.’ In: *CIDR*. Vol. 20. 2020, p. 29 (cited on pages 64, 68, 110, 111).
- [89] Moritz Sichert and Thomas Neumann. ‘User-defined operators: Efficiently integrating custom algorithms into modern databases’. In: *Proceedings of the VLDB Endowment* 15.5 (2022), pp. 1119–1131 (cited on pages 64, 67).
- [90] Kai Franz et al. ‘Dear User-Defined Functions, Inlining isn’t working out so great for us. Let’s try batching to make our relationship work. Sincerely, SQL’. In: *CIDR*. 2024 (cited on page 65).
- [91] Tim Fischer, Denis Hirn, and Torsten Grust. ‘Snakes on a plan: Compiling python functions into plain SQL queries’. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2389–2392 (cited on page 67).
- [92] Tim Fischer. ‘To Iterate Is Human, to Recurse Is Divine—Mapping Iterative Python to Recursive SQL’. In: *BTW 2023* (2023). doi: [10.18420/BTW2023-73](https://doi.org/10.18420/BTW2023-73) (cited on page 67).
- [93] Andrew Crotty et al. ‘Tuplware:” Big” Data, Big Analytics, Small Clusters.’ In: *CIDR*. 2015 (cited on page 68).
- [94] Andrew Crotty et al. ‘An Architecture for Compiling UDF-Centric Workflows’. In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1466–1477. doi: [10.14778/2824032.2824045](https://doi.org/10.14778/2824032.2824045) (cited on page 68).
- [95] Alfons Kemper and Thomas Neumann. ‘HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots’. In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 195–206 (cited on pages 69, 111).
- [96] John C. Reynolds. ‘The discoveries of continuations’. In: *Lisp and symbolic computation* 6 (1993), pp. 233–247 (cited on page 73).

- [97] Olivier Danvy and Lasse R Nielsen. ‘Defunctionalization at work’. In: *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. 2001, pp. 162–174 (cited on page 73).
- [98] Jeremy Gibbons. ‘Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity’. In: *arXiv preprint arXiv:2111.10413* (2021) (cited on page 73).
- [99] Ulrich Schöpp. ‘On interaction, continuations and defunctionalization’. In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2013, pp. 205–220 (cited on page 73).
- [100] Olivier Danvy. ‘Defunctionalized interpreters for programming languages’. In: *ACM Sigplan Notices* 43.9 (2008), pp. 131–142 (cited on page 73).
- [101] Yuri Gurevich. ‘Sequential abstract-state machines capture sequential algorithms’. In: *ACM Transactions on Computational Logic (TOCL)* 1.1 (2000), pp. 77–111 (cited on page 73).
- [102] C. Lawson. ‘How Functions can Wreck Performance’. In: *The Oracle Magician* IV.1 (Jan. 2005). <http://www.oraclemagician.com/mag/magic9.pdf> (cited on page 73).
- [103] R.W. Floyd. ‘Algorithm 97: Shortest Path’. In: *Communications of the ACM* 5.6 (1962) (cited on page 73).
- [104] Christian Duta and Torsten Grust. ‘Functional-Style SQL UDFs with a Capital ’F’’. In: *Proc. SIGMOD*. 2020 (cited on pages 74, 75, 91, 92, 104).
- [105] Christian Duta. ‘Another Way to Implement Complex Computations: Functional-Style SQL UDF’. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA ’22. Philadelphia, Pennsylvania: Association for Computing Machinery, 2022. DOI: [10.1145/3546930.3547508](https://doi.org/10.1145/3546930.3547508) (cited on pages 74, 103, 104).
- [106] *MySQL 8.0 Documentation*. <http://dev.mysql.com/doc/> (cited on page 75).
- [107] G.J. Sussmann and G.L. Steel. ‘Scheme: An Interpreter for Extended Lambda Calculus’. In: *AI Memo* 349 (1975) (cited on page 75).
- [108] Andrew W Appel. *Compiling with continuations*. Cambridge university press, 2007 (cited on page 81).
- [109] Andrew Kennedy. ‘Compiling with continuations, continued’. In: *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. 2007, pp. 177–190 (cited on page 81).
- [110] Youyou Cong et al. ‘Compiling with Continuations, or without? Whatever.’ In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–28 (cited on page 81).
- [111] Zoe Paraskevopoulou and Anvay Grover. ‘Compiling with continuations, correctly’. In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–29 (cited on page 81).
- [112] Marius Müller et al. ‘Back to Direct Style: Typed and Tight’. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023), pp. 848–875 (cited on page 81).
- [113] D. Michie. ‘“Memo” Functions and Machine Learning’. In: *Nature* 218.306 (Apr. 1968) (cited on page 88).
- [114] R.S. Bird. ‘Tabulation Techniques for Recursive Programs’. In: *ACM Computing Surveys* 12.4 (Dec. 1980) (cited on page 88).
- [115] G. Aranda et al. ‘R-SQL: An SQL Database System with Extended Recursion’. In: *Electronic Communications of the EASST* 64 (Sept. 2013) (cited on page 91).
- [116] Christian Duta. ‘Viability of Recursive SQL Functions’. PhD thesis. Universität Tübingen, 2022 (cited on page 104).
- [117] Frank Tip. ‘A survey of program slicing techniques’. In: *J. Program. Lang.* 3 (1994) (cited on page 104).
- [118] Mark Weiser. ‘Program Slicing’. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: [10.1109/TSE.1984.5010248](https://doi.org/10.1109/TSE.1984.5010248) (cited on page 104).

- [119] Torsten Grust, Nils Schweinsberg, and Alexander Ulrich. ‘Functions are data too: defunctionalization for PL/SQL’. In: *Proceedings of the VLDB Endowment* 6.12 (2013), pp. 1214–1217 (cited on page 105).
- [120] Torsten Grust and Alexander Ulrich. ‘First-class functions for first-order database engines’. In: *arXiv preprint arXiv:1308.0158* (2013) (cited on page 105).
- [121] Carsten Binnig et al. ‘FunSQL: It is Time to Make SQL Functional’. In: *Proceedings of the 2012 Joint EDBT/ICDT Workshops*. EDBT-ICDT ’12. Berlin, Germany: Association for Computing Machinery, 2012, pp. 41–46. doi: [10.1145/2320765.2320786](https://doi.org/10.1145/2320765.2320786) (cited on page 105).
- [122] Yanhong A Liu and Scott D Stoller. ‘From recursion to iteration: what are the optimizations?’ In: *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. 1999, pp. 73–82 (cited on page 106).
- [123] Yanhong A Liu, Scott D Stoller, and Tim Teitelbaum. ‘Static caching for incremental computation’. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.3 (1998), pp. 546–585 (cited on page 106).
- [124] Yanhong A Liu and Tim Teitelbaum. ‘Systematic derivation of incremental programs’. In: *Science of Computer Programming* 24.1 (1995), pp. 1–39 (cited on page 106).
- [125] Richard Bellman. ‘The theory of dynamic programming’. In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515 (cited on page 106).
- [126] Thomas Neumann and Alfons Kemper. ‘Unnesting arbitrary queries’. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015) (cited on page 110).
- [127] Marcus Huber. ‘Optimization of PL/pgSQL Translations Using Batching and Multiple Recursive References’. MA thesis. July 2022 (cited on page 110).
- [128] Timo Kersten, Viktor Leis, and Thomas Neumann. ‘Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra’. In: *The VLDB Journal* 30 (2021), pp. 883–905 (cited on page 111).

Special Terms

A

ANF administrative normal form. x, 5, 32–37, 47–50, 52–54, 63, 64, 80–85, 95

AST abstract syntax tree. 16, 29

C

CFG control flow graph. 29, 30, 33, 40

CPS continuation passing style. x, 7, 32, 73, 75, 81–85, 89, 97, 103–106

crude SSA is a trivial form of SSA where all possible ϕ -functions are inserted. 30, 33, 45, 47

CTE common table expression. vi, vii, 24, 25, 27, 28, 35, 37–43, 50, 53, 54, 59, 60, 63, 64, 68, 75–78, 81, 85–93, 97, 103–105, 109–111

D

DDL data definition language. 27

defunctionalization is a program transformation which eliminates higher-order functions. x, 7, 73, 75, 81–83, 85, 96, 97, 103, 105, 106

I

IPC inter-process communication system. 13

IR intermediate representation. 4–8, 29, 32, 63, 64, 68, 81, 95

R

RBAR row by agonizing row. 16

S

SPI server programming interface. 17–19

SSA static single assignment form. x, 5, 30–33, 37, 41, 45, 47, 48, 63–65, 78–81, 95, 96, 99, 105

T

TVF table-valued function. 16, 26, 39, 60, 67, 97

U

UDF user-defined function. v–vii, x, xi, 1–4, 8, 9, 15, 17, 19, 20, 23, 28, 29, 31–43, 45, 50, 53–60, 63, 65–68, 73–80, 82, 84–93, 95–101, 103–105, 109–111, 115, 116, 118, 120, 122, 124, 126–128

UDO user-defined operator. 67, 68

