

Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät

Wilhelm-Schickard-Institut für Informatik

Database Systems Research Group

Bachelor Thesis Computer Science

Bringing Flummi to umbra.trampoline

Markus Holder

08/31/2024

Reviewer

Prof. Dr. Torsten Grust

Department of Computer Science

University of Tübingen

Supervisors

Louisa Lambrecht

Department of Computer Science

University of Tübingen

Tim Fischer

Department of Computer Science

University of Tübingen

Holder, Markus

5447510

Bringing Flummi to umbra.trampoline

Bachelor Thesis Computer Science

Eberhard Karls Universität Tübingen

From 05/01/2024 to 08/31/2024

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor Thesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelor Thesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Markus Holder

Abstract

In modern database systems, iterative and recursive calculations in Python or C that use embedded Structured Query Language (SQL) queries often lead to performance issues due to the repeated entering and exiting of the database environment. This thesis addresses this problem by extending the "Flummi" compiler developed at the Database Systems Research Group from University of Tübingen, which can translate iterative and recursive input programs into pure SQL files.

The focus of this work was on adapting the compiler to generate output compatible with the **trampoline** feature available in the **Umbra** Database Management System (DBMS) and compiling it accordingly. This feature utilizes the principle of trampolining, inspired by functional languages such as Lisp or Haskell, which transforms iterative and recursive programs into loops with discrete calculations, enabling better parallelization and thus potentially significant performance gains.

A central aspect was the direct comparison of performance between the **umbra.trampoline** feature and the traditional implementation of iterative/recursive queries using Common Table Expressions (CTEs) in **Umbra**. The experiments showed that the trampoline feature offers significant advantages over CTEs, particularly in simpler and less complex recursive queries, by significantly reducing execution time and enabling more efficient parallelization. However, in more complex queries and larger datasets, the CTE-based implementation occasionally exhibited better performance, suggesting that the optimization potential of the trampoline feature is not yet fully realized.

Additionally, **Umbra** was compared with the **DuckDB** database system, where **Umbra** outperformed **DuckDB** in all test scenarios, especially in data-intensive tasks. These results highlight the efficiency of **Umbra** and the specific advantages of the trampoline approach, particularly in highly parallelized environments.

Contents

Abstract	v
Acronyms	ix
1. Introduction	1
2. Basics	3
2.1. The Umbra Database System	3
2.2. SQL	3
2.2.1. WITH [RECURSIVE]	4
2.2.2. Umbra and Trampolining	6
2.3. Python	11
3. Flummi compiler	13
3.1. Design and Architecture of the Compiler	14
3.2. Features and possibilities of Flummi	16
4. Implementation	17
4.1. Implementation Details of Individual Components	17
4.1.1. Expressions	19
4.1.2. Conditional expressions	19
4.1.3. Emits	20
4.1.4. Printing	21
4.2. Challenges and Solutions	23
4.2.1. CASE WHEN vs UNION ALL	23
4.2.2. Target table number	24
5. Experiments	25
5.1. WITH [RECURSIVE] vs. umbra.trampoline	26
5.2. Umbra vs. DuckDB	29
5.3. Discussion	31
6. Conclusion & Future Work	33
A. Appendix: UDF MUL	35
Bibliography	37

Acronyms

AST	Abstract Syntax Tree
CTE	Common Table Expression
CFG	Control Flow Graph
DCL	Database Control Language
DDL	Database Definition Language
DBMS	Database Management System
DML	Database Manipulation Language
IBM	International Business Machines Corporation
PL/SQL	Procedural Language/Structured Query Language
SSD	Solid State Drive
SQL	Structured Query Language
TCL	Transaction Control Language
TPC-H	Transaction Processing Performance Council Benchmark H
T-SQL	Transact-SQL
UDF	User-Defined Function

Introduction

A significant portion of database queries, especially those that are iterative or recursive, are often embedded in other languages. This separation leads to increased communication overhead between the application logic and the database system, which can impact efficiency. The compiler known as "Flummi" was developed to translate iterative programs from the research language PL/Flummi into SQL using CTEs such as `WITH RECURSIVE`. This process ensures that queries are executed entirely within the DBMS which significantly improves performance and reduces the need for external communication. In addition, the compiler is able to take into account the peculiarities of different SQL dialects and optimize them for specific systems, which increases the flexibility and adaptability of the system.

The goal of this research project is to extend the existing compiler to translate PL/Flummi input into the `umbra.trampoline` feature of the DBMS **Umbra**. This innovative technology represents a special variant of "trapolining", which makes it possible to represent iterative and recursive program steps in the form of loops with discrete steps. This not only significantly reduces the complexity of the queries, but also creates considerable potential for improved parallelization of the processes. Such improved parallelization could lead to a significant increase in processing speed, especially for complex and data-intensive operations. In addition, this technology opens up new possibilities to further optimize the efficiency and scalability of modern database systems and to adapt them to the growing demands of data processing.

Basics

This chapter provides an overview of the fundamental concepts and technologies relevant to this thesis. It covers the **Umbra** database system, essential SQL features including **WITH [RECURSIVE]** and the specific **umbra.trampoline** function, and introduces the Python programming language, which is integral to the development and implementation of the compiler.

2.1. The Umbra Database System

In recent years, database systems have had to balance performance and scalability. Traditional in-memory databases are fast but expensive and cannot handle large amounts of data. Disk-based systems are scalable but slow.

The **Umbra** database system [1] combines the best of both worlds: the speed of in-memory databases and the scalability of Solid State Drive (SSD)-based storage systems. This innovative approach uses the increasing capacities and decreasing costs of SSDs along with substantial in-memory buffers to achieve efficient and cost-effective data management. It employs a novel buffer manager with variable-size pages, which allows it to store large objects efficiently. This buffer manager handles both in-memory and disk-based data. It maintains high performance when the working set fits in memory and efficiently manages accesses to SSDs when data exceeds memory capacity.

Furthermore, **Umbra** uses an adaptive compilation strategy to balance compilation and execution times. It enables fine-grained execution control and efficient parallelism, ensuring optimal performance across a variety of workloads.

The system outperforms HyPer and MonetDB in various scenarios [1]. It is suitable for modern data-intensive applications where both cost and performance are critical. **Umbra** represents a significant advancement in DBMS.

2.2. SQL

SQL is a standardized language for managing and manipulating data in relational database systems [2]. Its origins can be traced back to the early 1970s, when it was developed by International Business Machines Corporation (IBM) under the name SEQUEL (Structured English Query Language). The name was later changed to SQL for trademark reasons. Since then, SQL has become the central technology for accessing relational databases and is now an essential component of database management and analysis.

SQL statements can be divided into several categories: Database Manipulation Language

(DML), Database Definition Language (DDL), Database Control Language (DCL), and Transaction Control Language (TCL).

The DML is used to manipulate data in a relational database. It includes statements such as **SELECT**, **INSERT**, **UPDATE** and **DELETE**. These commands make it possible to retrieve, insert, update and delete data, allowing the data stored in the database to be manipulated, efficiently managed and used. For example, **SELECT** is used to retrieve data from one or more tables, while **INSERT** inserts new records into a table.

The DDL is used to define and manage the structure of a database. Statements such as **CREATE**, **ALTER** and **DROP** can be used to create new databases, tables and other database objects, change existing structures and delete objects that are no longer required. This sub-language ensures that the database structure is clearly defined and organized, enabling efficient storage and management of data. A typical example is **CREATE TABLE**, which is used to create a new table with a specific schema.

In addition to DML and DDL, there is the DCL to control access to the database and the TCL to manage transactions within the database.

Moreover, various DBMS have specific extensions and additional features that extend beyond the capabilities of standard SQL. These extensions enable special functions that are useful for certain use cases or performance optimizations. For example, Oracle offers special Procedural Language/Structured Query Language (PL/SQL) extensions for procedural programming, while Microsoft SQL servers use Transact-SQL (T-SQL), which includes additional control structures and error handling mechanisms. This thesis will primarily concentrate on the **Umbra**-specific extension **umbra.trampoline**, which is currently only accessible in this format within **Umbra**.

2.2.1. WITH [RECURSIVE]

CTEs are a significant enhancement to the SQL language, primarily aimed at improving the readability and maintainability of queries. By utilizing CTEs, temporary result sets can be defined, which can be referenced multiple times within the overall query. These temporary result sets facilitate writing complex queries and promote a clearer and more structured representation of the query process.

A particularly noteworthy form of CTEs is the recursive CTE, introduced in the SQL-99 [3] standard through the keyword **WITH RECURSIVE**. Recursive ones extend the basic concept of CTEs by allowing a query to reference itself. This self-referencing capability is invaluable, especially when dealing with hierarchical or recursive data structures. Examples include organizational hierarchies, family tree structures, or networks of nodes and connections.

A recursive CTE consists of three essential components:

- **Initial Query:**

This query defines the starting point of the recursion. It is executed only once and establishes the initial set of records that serve as the basis for recursive processing.

- **Recursive Query:**

This component references the results of the initial query as well as itself. It is executed repeatedly, progressively expanding the result set by adding additional records that meet the recursion conditions.

- **UNION ALL:**

This keyword links the results of the initial query and the recursive query. Through this linkage, all intermediate results are consolidated into a single result set.

The recursive process continues until no new records are found that satisfy the recursion conditions. This ensures that the recursion terminates once the entire hierarchical or recursive data model has been fully queried.

At the end of the recursion process, a final query is typically performed to present the ultimate results from the recursive CTE. This concluding query can be used to further filter, sort, or format the data to meet specific analysis or reporting requirements.

On [Listing 2.1](#), the pseudocode of a factorial function is presented to illustrate the corresponding process. An example has been prepared to demonstrate the calculation of the factorials for a given number *n*. The algorithm takes *n* as input and returns the factorial of every integer up to *n*.

```
1 i <- 1
2 val <- 1
3 n <- input
4
5 WHILE i <= n DO
6     PRINT n, val
7     i = i + 1
8     val = val * i
9 END
```

Listing 2.1: Pseudocode of factorial function

In the case of an implementation with **WITH RECURSIVE**, our example for calculating the factorial would look like [Listing 2.2](#):

```
1 WITH RECURSIVE fac(i, val) AS (
2     SELECT 1 AS i, 1 AS val
3
4     UNION ALL
5
6     SELECT i + 1, val * (i + 1)
7     FROM fac
8     WHERE i < input
9 )
10 SELECT fac.val FROM fac;
```

Listing 2.2: **WITH RECURSIVE** code in SQL

While recursive CTEs offer a powerful tool for writing complex recursive queries, they come with certain challenges that limit their widespread use. The semantic complexity of recursive queries can make them difficult to understand and implement correctly [4]. Additionally, the performance of queries using `WITH RECURSIVE` can be suboptimal, particularly with large datasets or deep recursion levels.

In summary, while recursive CTEs provide a valuable tool for certain types of data analysis and processing, their practical use is often constrained by their complexity and performance limitations. Careful consideration and testing are essential to determine the most appropriate approach for each specific use case.

2.2.2. Umbra and Trampolining

Trampolining [5] is an advanced technique in programming that aims to design iterative and recursive programs in such a way that they do not build up a deep recursion chain. Instead, a loop is used to handle the iteration/recursion efficiently. This method can be particularly helpful in preventing stack overflow, which is often caused by recursions that are too deep. Avoiding deep recursion can also improve the performance of the application. As a rule, trampolining is widely used in functional programming languages such as Lisp, Scheme or Haskell, which often work with iterative and recursive algorithms.

In SQL, trampolining provides an alternative to the traditional `WITH [RECURSIVE]`-clause by breaking the computations into discrete steps managed by a central controller, which can significantly improve efficiency and parallelizability.

2.2.2.1. Semantics

In trampolining, the program is organized in a central control loop, the so-called trampoline operator. This loop manages the control flow and ensures that each calculation step is executed in sequence. Instead of functions making direct recursive calls, control returns to the central loop after each step, which then decides how to proceed. In this way, the entire calculation is transformed into a kind of state machine, which can be combined particularly well with SQLs iterative constructs such as recursive CTEs.

By circumventing the CTEs, as traditionally used in SQL, the code becomes less deeply nested and thus avoids some of the typical problems of iterative/recursive implementations. As previously discussed in [6] and [7], the use of trampolining enables more efficient handling of computations, particularly in complex use cases. The semantics of trampolining allow for the representation of loops and branches in a way that not only simplifies the code, but also enables parallelizable computations. This is particularly useful in data-intensive applications where large amounts of data need to be processed. The use of trampolining can lead to significant performance improvements in such cases, as it supports efficient parallel implementation within modern database systems.

An essential feature of trampolining is its capacity to facilitate iterative computation in a modular and extensible manner, a capability that is frequently unavailable with recursive CTEs. While recursive CTEs are constrained by specific syntactic requirements and frequently encounter performance limitations, trampolining provides a versatile alternative that integrates seamlessly into existing SQL theories and practices. The independent assessment of each phase in the trampolining process also fosters high parallelism and efficiency (see [Listing 2.3](#)).

```

1 WITH TRAMPOLINE
2 T(b, c_1, ..., c_m) BRANCH(b) AS (
3   BRANCH 0: q_0
4   BRANCH 1: q_1(T)
5   ...
6   BRANCH n: q_n(T)
7 )
8 TABLE T;
9
10 W ← {q_0}
11 U ← W[b = 0]
12
13 REPEAT
14   I ← ∅
15   FOR t ∈ {1, ..., n}
16     FOR w ∈ W[b = t]
17       L ← I ∪ {q_t(w)}
18   U ← U ∪ U[b = 0]
19   W ← I[b ≠ 0]
20 UNTIL W = ∅
21 RETURN U

```

This section is ideal for parallel processing, as the structure of the code allows several processes to be executed simultaneously.

Listing 2.3: Semantik of Trampoline, adapted from [6]

The DBMS **Umbra** offers a functionality that enables this technique, which is remarkable as it is not included in the current SQL standard. Usually, recursive queries in SQL are implemented by a CTE that uses the **WITH RECURSIVE** statement. However, this method is often difficult to write and understand due to its complex and non-intuitive syntax [4]. In addition, it can be inefficient with deep recursions and put a heavy strain on system resources. **Umbra** provides an alternative approach: Instead of using recursive CTEs, **Umbra** allows the use of specialized control structures.

2.2.2.2. Syntax

`umbra.trampoline` has a number of syntactical subtleties that must be taken into account when using it. These subtleties can be divided into several steps, which are explained in detail below:

- **Call in normal query:**

In order to utilise the trampoline function, it must be included in the **FROM** clause. Its behaviour is analogous to that of any other call in this context. Consequently, it is possible to assign it an alias or to call several instances of it or of other tables in succession. The structure is in ascending order, whereby the tables are numbered. The initial table is always numbered with **0**, the subsequent tables with **1** and so on. Should one wish to access a specific table at a later stage, it is necessary to specify the corresponding number in the target. To access the third table created, for example, one must use the integer **2** in the **SELECT** command (see [Listing 2.4](#)).

```
1 SELECT *
2 FROM umbra.trampoline(
3     TABLE(...),           --Table number 0
4     TABLE(...),           --Table number 1
5     (...),
6     TABLE(...)            --Table number n-1
7 ) AS alias;
```

Listing 2.4: Call of `umbra.trampoline`

- **Initialization table:**

The first table serves as initialization table and contains all relevant and used columns. In all tables the first column serves as target input for the trampoline "jump", mean there is an integer that indicates which table should be processed next. The select otherwise behaves like a normal query **SELECT**, so the values can be initialized from a **FROM** clause or passed directly. Only the initialization table does not require a **FROM** reference to trampoline. Otherwise, the call behaves like a normal query with nested subqueries. It is important to note that columns that are not used from the start must still be initialized, in this case with the **NULL** value. All columns used in the trampoline feature must be declared from the start (see [Listing 2.5](#)).

```
1 TABLE(SELECT TargetTableNumber ,
2         column_1 AS col_name_1,
3         (NULL AS (column_2 type)) AS col_name_2,
4         (...),
5         column_n AS col_name_n
6     [FROM from_item_1[, (... from_item_m]]
7 )
```

Listing 2.5: Initialize table

- **Trampoline step:**

The individual trampoline steps are responsible for the actual processing of the feature. This is where the decision is made regarding which step is to be executed next, i.e. which table is to be accessed and its actual execution. All standard SQL operations, including CTEs and subqueries, can be used. The first column serves as the "targetTable" input for the next step. This column can assume various values that are defined by expressions, for example by a **CASE WHEN** construct or by means of **UNION ALL**. A single path is followed, which specifies the next step. It is important to note that a trampoline step table must always refer to trampoline in the **FROM** clause (but also to other tables or subqueries). [Listing 2.6](#) gives an example:

```
1 TABLE(SELECT CASE WHEN condition_1 THEN targetTableNumber_1
2             (...)
3             WHEN condition_k THEN targetTableNumber_n
4             ELSE targetTableNumber_n END,
5             expression_1 AS Col_name_1,
6             (...)
7             expression_n AS col_name_n
8             FROM trampoline [, (...) from_item_n]
9 )
```

Listing 2.6: Trampoline step with **CASE WHEN**

- **Emitting:**

In order to receive an output, it is necessary to select the number **0** as the target. The feature then outputs a value to the result set. If several outputs are required in one call of the feature (i.e. an independent table), this must be solved with **UNION ALL**. This involves jumping to **0** on one side and to another target table that is not **0** on the other side. In this instance, it is imperative to impose a **WHERE** clause on the query in order to establish a termination condition. Otherwise, the feature will result in an infinite loop (see [Listing 2.7](#) for clarification).

```
1 TABLE(SELECT 0,
2             column_1,
3             (...)
4             column_n
5             FROM trampoline
6             UNION ALL
7             SELECT targetTableNumber,
8             column_1,
9             (...)
10            column_n
11            FROM trampoline
12            WHERE condition
13 )
```

Listing 2.7: Emitting

To demonstrate how the `umbra.trampoline` works, we use our example of the factorial function. The individual trampoline parts required are shown in Listing 2.8:

```
1 SELECT fac.val
2 FROM umbra.trampoline(
3     TABLE(SELECT 1,
4              1 AS i,
5              1 AS val
6            ),
7     TABLE(SELECT 0,
8              i,
9              val
10            FROM trampoline
11           )
12     UNION ALL
13     SELECT 1,
14           i + 1 AS i,
15           val * (i + 1) AS val
16     FROM trampoline
17     WHERE i < input
18 )
19 ) AS fac;
```

Listing 2.8: `umbra.trampoline` code of factorial function

In conclusion, trampolining represents an elegant solution to the problem of deep recursion. The conversion of recursive calls into iterative loops not only eliminates the risk of very deep recursive nesting but also improves performance through parallel processing. The DBMS **Umbra** employs this technology in SQL and exemplifies how traditional programming problems can be solved with modern approaches. Furthermore, these approaches are much easier for humans to read and create, as they have a more intuitive structure.

2.3. Python

Python is a widely used [8], interpreted programming language that is known for its simplicity and readability. It was developed by Guido van Rossum in 1991 [9] and is ideal for fast development cycles and complex applications. **Python** is characterized by a clear syntax and an extensive standard library with many ready-made modules and functions. It is platform-independent and runs on Windows, macOS, and Linux. The interactive nature of Python facilitates testing and debugging.

The extensive and active community provides a wealth of resources that facilitate learning and problem-solving. Python supports various programming paradigms, enabling the implementation of complex compiler logic. Consequently, the performance weaknesses of **Python** have a relatively minimal impact on this type of compiler.

The Flummi compiler and its extension are written in **Python**, and therefore the extension for the `umbra.trampoline` output was also written in this language.

3

Flummi compiler

This chapter describes the Flummi compiler [10], developed by the Database Systems Research Group at the University of Tübingen. This compiler translates PL/Flummi (see Listing 3.1), a scientific language, into raw SQL statements. It is designed to facilitate the processing of complex logical sequences in relational databases. The main features of the compiler include source code parsing, semantic analysis, intermediate representation generation, optimizations, code generation, and code interpretation.

This section provides a detailed overview of the design, architecture, modules and components, control flow mechanisms and possibilities offered by the compiler.

```
1 CALL ($4$, $5$) IN
2 FUN (a: $int$, b: $int$) -> $int$: {
3   c: $int$;
4
5   c <- $0$;
6
7   LOOP multiplication_loop {
8     c <- ${0} + {1}$[a, c];
9
10    EMIT ${0}$[c];
11
12    b <- ${0} - 1$[b];
13
14    IF $0 = {0}$[b]
15      THEN BREAK multiplication_loop
16      ELSE CONTINUE multiplication_loop
17  };
18
19  STOP
20 }
```

Listing 3.1: PL/Flummi code of User-Defined Function (UDF) `mul`

3.1. Design and Architecture of the Compiler

The Flummi compiler is designed with a highly modular architecture, which enables the clear separation of concerns. This facilitates both maintenance and extensibility. One of the fundamental elements of the compiler is the parser, which is situated within the `parser.py` module. The parser is tasked with the conversion of the source code into an Abstract Syntax Tree (AST) (see Figure 3.1). The process starts with the decomposition of the source code into tokens, a task performed by the `tokenize` function, which employs a regular expression to identify the various token types. Subsequently, the parser applies grammatical rules to the aforementioned tokens in order to construct the AST, which is a hierarchical representation of the source code that captures its syntactic structure. This parsing process is orchestrated by the `parse` function, which systematically interprets tokens in accordance with the rules defined in the grammar, thereby ensuring that the resulting AST accurately reflects the logical flow of the program. The AST serves as the foundation for subsequent stages of the compilation process, including analysis, optimization, and code generation.

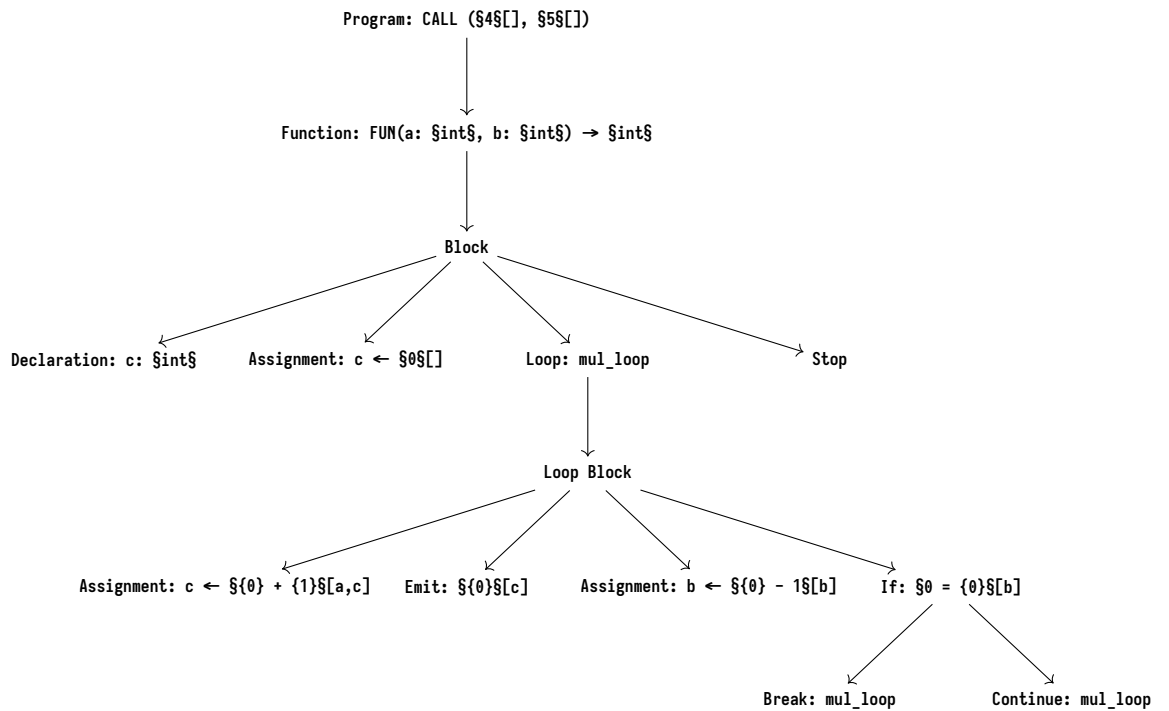


Figure 3.1.: Abstract Syntax Tree for the UDF `mul`

In the subsequent phase, the analyzer performs a comprehensive semantic analysis of the AST to guarantee the accurate declaration of all variables and the consistent and appropriate usage of data types. This essential analysis is conducted by the `analyzer.py` module. During this phase, the analyzer constructs a symbol table, which maps each variable to its corresponding type, ensuring that the program adheres to the expected type constraints. This symbol table plays a significant role in verifying the correctness of variable usage throughout the code.

Upon completion of the semantic analysis, the AST is transformed into an intermediate representation, also referred to as a Control Flow Graph (CFG), as illustrated in Figure 3.2. This conversion is managed by the `lowering.py` module. The lower function systematically breaks down complex programming constructs into finer components and translates them into simple blocks and control flow structures, including `GOTO` and `JUMP` statements. These structures differ in that a `GOTO` interacts with upstream or downstream blocks, while a `JUMP` forwards the data directly to a specific target block.

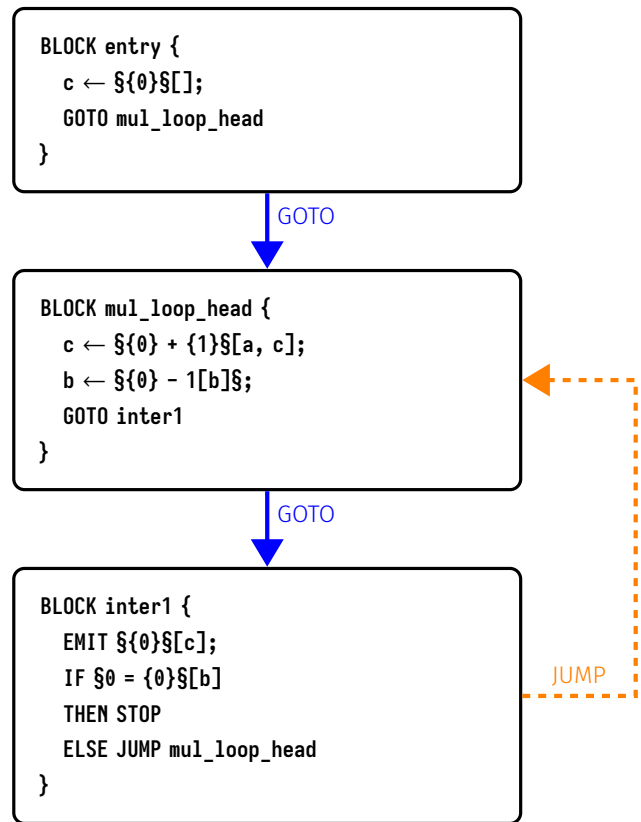


Figure 3.2.: Control Flow Graph of UDF mul

The optimizer enhances the efficiency of the generated code by employing a multitude of optimization techniques. One of the principal techniques employed is the elimination of unreachable code, which removes portions of the code that are incapable of being executed, thereby enhancing the program's efficiency. Furthermore, the optimizer inlines functions, integrating the function body directly into the calling context to reduce overhead and improve execution speed. These optimizations are carried out within the `optimizer.py` module, which works by analyzing the CFG to identify and apply these improvements, ensuring that the resulting program is both efficient and optimized for performance.

Once the CFG has been optimized, it is passed to the code generator, where it is translated into raw SQL statements. This step is performed by the `codegen.py` module, which meticulously converts the optimized CFG into SQL code that can be executed by the target database system. The modular design of the Flummi compiler ensures that each stage of the compilation process — from parsing to code generation — is clearly separated. This delineation not only facilitates comprehension and maintenance of each phase but also allows each component to be tested and refined independently, resulting in a more robust and flexible compiler.

3.2. Features and possibilities of Flummi

The Flummi compiler can be invoked from the console and offers a variety of options for generating customized output, which can be controlled by using additional flags when running the compiler. These flags allow users to customize the compilation process to meet specific requirements, particularly with regard to the target DBMS. This offers considerable flexibility, as different database systems have different requirements and capabilities that must be taken into account during compilation.

A key feature of the Flummi compiler is its capacity to designate a particular DBMS for which the compiler executes tailored optimizations. An illustrative example is the `materialize` feature. This feature is automatically applied to all CTE that are referenced more than once within a query. This results in enhanced performance by circumventing the repetition of calculations for the same data set. However, this feature is only enabled if the pertinent DBMS offers support for it, as not all database systems provide this optimization option.

In addition to the DBMS-specific flags, the Flummi compiler also provides general flags that control the entire process of lowering and optimization. These flags have an impact on certain operations performed during compilation and allow users to further refine the compilation process. One such flag is the `JUMPS_ONLY` flag, which, if enabled, converts all `GOTO` transitions between blocks into `JUMP` commands.

Additionally, other flags may be employed to activate or deactivate supplementary functions, depending on the particular specifications of the user. For instance, the generation of `traces` may be initiated to procure comprehensive data regarding the program's execution, which can be especially advantageous in intricate analysis or debugging scenarios.

These flags let the Flummi compiler adapt the generated output to the user's needs and the target DBMS capabilities. This makes it flexible and efficient to use, in development and in production. Users can optimize their applications for their target environment to get the best performance.

Implementation

To implement and test the `umbra.trampoline` feature, a nearly fully functional version of **Umbra** was required. This version was obtained from the Technical University of Munich [11]. The Linux-based version of **Umbra** provided the necessary environment to test the code generated by the Flummi compiler, identify errors, and correct them in a step-by-step manner. With the provided version, it was possible to load the Transaction Processing Performance Council Benchmark H (TPC-H) database and test various UDFs, ensuring that the system behaved as expected. For illustrative purposes, [Appendix A](#) contains an `umbra.trampoline` code generated by the modified Flummi version of UDF `mul`.

Additionally, a functional **Python** installation was needed on the computer to run the Flummi compiler. It was crucial to ensure that the **Python** version was not below 3.12 to avoid compatibility issues. All extensions made to the compiler were written in **Python**.

The remainder of this chapter will delve into the details of how, why, and where we made the most significant changes. We will discuss the challenges encountered during the development and integration of the new features and the solutions that led to a successful implementation, highlighting the improvements made to the system's functionality and efficiency.

4.1. Implementation Details of Individual Components

In the original Flummi compiler, the imperative code is converted into a CFG, as detailed in [section 3.1](#). This step is essential, as it transforms the code into a structure that can be utilized to generate `umbra.trampoline`-compatible code. In the lowering phase, as previously described, superfluous blocks are removed from the CFG to facilitate the most efficient path. Upon completion of this process, a CFG is generated, wherein each block can be translated into a table in `umbra.trampoline`. The utilization of the `JUMPS_ONLY` flag proved instrumental in this regard, as every transition between blocks in `umbra.trampoline` is exclusively realized by a `JUMP`, rather than a `GOTO`. This is because the data is organized by the trampoline operator, which determines the target blocks. It should be noted that these are not only dependent on upstream or downstream tables. This flag effectively replaced all `GOTO` statements with a `JUMP`, thereby facilitating the successful translation of the resulting CFG into `umbra.trampoline` tables.

To generate `umbra.trampoline` code from PL/Flummi code using the Flummi compiler, an additional flag, `WITH_TRAMPOLINE`, has been introduced. In conjunction with the **Umbra** database flag, this flag guarantees the generation of `umbra.trampoline` code. However, if a different database is selected and the `WITH_TRAMPOLINE` flag is enabled, this flag is ignored and the original `WITH [RECURSIVE]` code is generated. Conversely, if the `WITH_TRAMPOLINE` flag is not enabled, only the

original `WITH [RECURSIVE]` code is generated, even when using `Umbra`. This prevents inadvertent compilation to trampoline code, which is only functional in `Umbra`.

The semantic rules that apply to Flummi with CTE constructs are identical to those for `umbra.trampoline`. This means that all properties and behaviors specified for Flummi can also be seamlessly transferred to `umbra.trampoline`. Thanks to this consistency, the constructs of Flummi can be easily adopted and adapted.

The most significant challenge was to precisely categorize the individual code blocks, as this required the use of different methodological approaches to represent them as correctly executable tables. These blocks can contain different statements, such as if conditions, emits (outputs) or expressions that correspond to variable changes within the imperative code. It is also possible for these statements to occur in combination, for example in a block that contains both an emit and an if condition. This section details the specific methods and techniques that have been developed and applied to handle these different cases and ensure that each block is processed correctly.

To understand how to distinguish and handle the different cases, it is necessary to provide a brief overview of the compiler's structure, particularly how it processes, stores, and utilizes various values. In this context, we will focus on three key aspects:

- **Labels:**

Labels are stored as instances of the `BlockLabel` class, each containing a unique string identifier. These labels identify the blocks within the CFG of the program. The CFG is stored in a **Graph**, which consists of an `entry_label` marking the starting point of the program and a dictionary (`blocks`) that contains all the blocks of the CFG. Each block is associated with a specific `BlockLabel`.

- **Graph:**

The entire CFG of a program is stored in a `Graph` object. The **Graph** object plays a crucial role in representing the program, as it encapsulates the control flow structure in a compact and accessible form.

- **Block:**

A block in a CFG represents a sequence of statements that are executed without interruption, e.g. by a `JUMP`. Each block contains a label that uniquely identifies it, a list of statements that represent the instructions of the block as **statements** objects (including instructions such as `Emit`, `Assignment`, `If`, etc.), and a terminal that marks the end of the block and typically describes a control flow change, such as a `JUMP` or `GOTO`.

- **VariableBindings:**

The mapping between variables (`Variable`) and their associated expressions (`Expression`) is stored in a dictionary called `VariableBindings`. In this dictionary, each variable acts as a key, while the corresponding expression is stored as the value. These bindings are essential for the compiler, as they enable the substitution of variables with the correct values or expressions during code generation.

By understanding these core components it becomes clear how the compiler structures, processes, and ultimately translates the various elements of a program into executable code.

4.1.1. Expressions

In the implementation of `umbra.trampoline`, expressions play an essential role, as they represent the computations or transformations applied to variables. Each variable within a block is associated with a specific expression (or the value itself), which dictates how the value of that variable is calculated in the generated SQL code. It is essential that these expressions are correctly mapped and integrated into the generated SQL to ensure that data processing within the **Umbra** environment operates correctly.

The management of these mappings is handled by the `VariableBindings` dictionary, which links each variable to its corresponding expression. During the SQL code generation process for `umbra.trampoline`, these expressions are assigned to the appropriate columns, ensuring that the resulting SQL code executes the intended computations. Proper initialization of all variables and accurate recognition of their types are critical to avoid runtime errors and maintain the integrity of the data processing.

After the variables have been initialized in the `Initalize` table with either the values specified in the PL/Flummi input or `NULL`, the corresponding columns can now be accessed, and the expressions defined in the block can be applied to them. Finally, the values are cast to the appropriate data type. This information is used to ensure that the generated SQL code adheres to the correct data types, to avoid runtime errors and to ensure that the SQL queries generated by the compiler are both syntactically correct and semantically meaningful.

4.1.2. Conditional expressions

In the event that a CFG block contains a condition, there are at least two options with regard to which block must be forwarded. The determination of this decision is ultimately contingent upon the respective conditions. It is possible that a block may contain a plurality of these conditions. In a typical scenario, these conditions are organized in an "if-else" structure, branching to block **A** if the condition is met and to block **B** otherwise. If there are several "if" conditions, these can also be nested.

In order to create a variant that is as efficient and easy to read as possible, we have opted for the `CASE WHEN` construct [12]. This construct is similar to a classic "if-else" condition in that it receives a condition and, if this is fulfilled, selects path **A**, otherwise path **B**. The advantage of the `CASE WHEN` construct is that any number of conditional cases can be strung together, which avoids deep and difficult-to-understand nesting.

Nevertheless, a specific challenge emerges from the fact that in the absence of any applicable conditions, a `NULL` value is returned. This results in an invalid command within the branch expression of `umbra.trampoline`. To circumvent this issue, we have ensured that a value is returned in all potential combinations. This was accomplished by enumerating all conditions within the `WHERE` clause, thereby ensuring that only those lines are considered that possess a valid return value. (see [Listing 4.1](#))

Our initial concerns that this procedure could lead to a loss of performance due to the need for additional filtering have not been confirmed. **Umbra's** optimizer works

very efficiently in this context. However, the introduction of this clause was necessary to ensure that all possible inputs can be compiled correctly and completely.

```
1 TABLE(SELECT CASE WHEN Condition_1 THEN TargetTableNumber_1
2                   WHEN Condition_2 THEN TargetTableNumber_2
3                   (...)
4                   WHEN Condition_n THEN TargetTableNumber_n
5                   END,
6                   COL_1,
7                   COL_2,
8                   (...),
9                   COL_n
10          FROM      trampoline
11          WHERE     Condition_1
12                  OR Condition_2
13                  (...)
14                  OR Condition_n
15 )
```

only returns the rows that result in a valid value

Listing 4.1: Conditional expressions in `umbra.trampoline`

4.1.3. Emits

Emits indicate that a value should be added to the final output. As described in 2.2.2.2, a table with `targetTableNumber = 0` in the first `SELECT`-clause column must be created for this purpose, which will supplement the final result set. These emissions can occur under various conditions in a CFG-block. In order to differentiate between these conditions and to modify the compiler output accordingly, it was necessary to ascertain the membership of the emits. This was achieved by applying the `isinstance()` method of Python to the block. The affiliation is limited to the following three scenarios:

- **Raw Emit:**
The specified block contains only an `Statement Emit` and no subsequent blocks or expressions. In this case, the sole purpose of the block is to ensure that the value is added directly to the final output without any additional processing or conditions.
- **Emit with Expression:**
In this context, the `Statement Emit` is linked to a more intricate expression that may entail calculations or data transformations prior to their emission. This scenario is prevalent in instances where the output needs some form of preprocessing before being incorporated into the final result set.
- **Emit with Assignment:**
In the event that a block contains both an emit instruction and a forwarding (assignment), it is imperative to consider both scenarios. On the one hand, the emission of the value and, on the other hand, the forwarding of the control flow to the subsequent block. In this case, the compiler generates two paths: one that carries out the emission of the value and

another that forwards the control flow accordingly (see Listing 4.2). The forwarding path can itself be provided with further conditional expressions in order to map more complex control logic within the CFG.

```
1 TABLE(SELECT 0,
2     col_1,
3     col_2,
4     (...),
5     col_n,
6     CAST(("col_x")) AS outputType) AS "%result%"
7 FROM trampoline
8
9     UNION ALL
10
11 SELECT TargetTableNumber,
12     Expression_1,
13     Expression_2,
14     (...),
15     Expression_n,
16     FROM trampoline
17 )
```

Listing 4.2: Emit with Assignment in `umbra.trampoline` syntax

4.1.4. Printing

In the context of the Flummi compiler, "printing" refers to the formatting of the output code in a manner that facilitates human readability. This is essential for effective debugging, analysis of optimizations, and assurance that alterations do not alter the fundamental logic of the code.

As SQL syntax permits the composition of all statements on a single line [13], this could prove challenging for humans to interpret and debug, particularly in the absence of whitespace sensitivity, which can impede comprehension. For this reason, we have made a conscious decision to design the output in such a way that it is readily comprehensible through clear structuring and the deliberate use of line breaks. In doing so, we have adhered to some basic principles to ensure that the code is not only correct, but also easily accessible for people.

- **Tables**

In order to achieve the desired output, it was necessary to adapt the `codegen.py` file and implement the desired operations therein. The resulting string corresponds to the desired SQL syntax and is generated in a manner similar to that of the original Flummi compiler. As outlined in subsection 2.2.2, the initialisation of all variables utilised within the PL/Flummi input is contingent upon the creation of an initialisation table. The aforementioned requirement has been implemented in such a way that an initialization table is generated subsequent to the invocation of the function. This initialization table initializes all unassigned variables with either a start value or `NULL`. The corresponding variables

may be derived directly from the `variable_bindings` list generated by the Flummi compiler. Furthermore, a column designated `"%result%"` with the value `NULL` has been incorporated, which corresponds to the output type.

Subsequently, all blocks within the CFG class were enumerated in a loop. The type (see [subsection 4.1.1](#) until [subsection 4.1.3](#)) of each block was verified through the application of a series of conditions, which were encapsulated in a distinct method. This method generates the corresponding blocks and also returns them as a string.

As a result of these modifications, all tables were constructed in accordance with the `umbra.trampoline` syntax.

- **Line Breaks**

In order to facilitate comprehension of the code, we have elected to transmit a single piece of information in each line. This implies that each line contains a single instruction of the code. For instance, the columns in the `SELECT` clause have been transferred to a distinct line. In the event that one of these lines incorporates an expression, this is similarly displayed on the same line. In the event that the expression contains subqueries, these were replicated in their original form from the input file (PL/Flummi). This approach has enabled us to guarantee that the code remains both readable and comprehensible for humans.

The implementation was achieved through the utilization of the `string.join` construct [14], a Python-specific method that enables the concatenation of character strings with line breaks, beginning with the second value.

- **Label and table number**

To further increase the comprehensibility of the code, we have added a comment line for each table in addition to the necessary code sections, indicating both the table label and the table number. These comments serve as an orientation aid by making clear the context of the respective tables within the code.

This addition makes it much easier to understand the flow of the code and the corresponding jumps between the tables. Especially when working with complex structures, these comments facilitate understanding and enable faster error analysis and correction.

```
1 -- Label: label_1 Table number: 1
2 TABLE (...)
3 -- Label: label_2 Table number: 2
4 TABLE (...)
5 (...)
6 (...)
7 -- Label: label_n Table number: n
8 TABLE (...)
```

Listing 4.3: Comments between the tables

4.2. Challenges and Solutions

Not all of the solutions we initially conceived were successful. Some ideas and implementations had to be discarded and revised. Certain problems also arose due to the structure of the Flummi compiler and its way of processing and outputting data. In the end, we had to make compromises between feasibility, performance, and aesthetics in some areas.

4.2.1. CASE WHEN vs UNION ALL

Originally, we planned to implement the condition branches using a **UNION ALL** variant. This approach involved prefixing each table containing a branch with a CTE to perform the calculations, followed by a subquery for all possible paths using **UNION ALL**. Since only one of these paths would actually yield a result, this method is semantically equivalent to the **CASE WHEN** variant. This would have had the following syntax like in [Listing 4.4](#):

```
1 TABLE(WITH "%assign%" (COL_1,COL_2, (...) COL_n) AS (  
2     SELECT expression_1 AS Col_name_1,  
3         (...)  
4         expression_n AS col_name_n  
5     FROM trampoline  
6 )  
7 SELECT targetTableNumber_1, COL_1,COL_2, (...) COL_n  
8 FROM "%assign%"  
9 WHERE condition_1  
10  
11 UNION ALL  
12  
13 SELECT targetTableNumber_2, COL_1,COL_2, (...) COL_n  
14 FROM "%assign%"  
15 WHERE condition_2  
16  
17 (...)  
18  
19 SELECT targetTableNumber_n, COL_1,COL_2, (...) COL_n  
20 FROM "%assign%"  
21 WHERE condition_n  
22 )
```

Listing 4.4: Conditional expressions with CTE and **UNION ALL**

Nonetheless, we have identified significant performance issues that resulted in considerable delays in executing certain queries. This could potentially be attributed to the fact that each subquery was executed as a complete query individually, and these were not optimized away during execution. In contrast, when using the **CASE WHEN** construct, the query is executed only once for the respective case.

4.2.2. Target table number

In the trampoline feature, the jumps are specified by integers located in the first column of the trampoline-table. We need to make sure that the correct paths are taken. Our CFG has a successor block for each block we want to use as a target table. However, since these are given specific Labels by Flummi, such as `loop_head`, `inter1`, etc., we need to map them chronologically to a number that we can later specify as the target table.

In Python, this is relatively easy to do using the `enumerate()` function, which returns the index of each element in a list as a number. You can create a dictionary from the list of targets and use it to assign a unique integer to the targets provided by Flummi. In the assignment part of our compiler, we just only need to look up the target name in the dictionary and the corresponding integer value will be returned.

Experiments

To test the performance differences, a comparison was made between `WITH [RECURSIVE]` and `umbra.trampoline` in Umbra. To get an estimation for the performance, a comparison between Umbra and DuckDB was also conducted. All benchmarks were performed on an AMD RYZEN 5 3600XT CPU with 6 physical and 12 logical cores running at 4 GHz. The system has 36 GB of main memory and all database files are stored on a SAMSUNG 970 EVO SSD with 1 TB of storage.

Benchmarking databases is not easy, and it is a challenge to make it fair and meaningful [15]. In this case, we tried to create a repeatable test environment that was as comprehensive and varied as possible. Umbra was tested in a pre-release version with a buffer size of 16 GB (standard 50 percent of the machine). DuckDB [16] was tested in version 1.0.0 with a buffer size of 26 GB, which is the default (80 percent). The default values were used intentionally, although both database systems could be optimized, for example by ensuring an optimal thread memory ratio in DuckDB [17]. However, for the sake of reproducibility, comparisons were made using the default settings.

For comparison, we used several UDFs, some of which are based on the TPC-H database benchmark [18] with a scale of 1. We also chose UDFs that do not require any special setup. Unfortunately, since Umbra is still under development, some features are not yet implemented. This limited our choice of UDFs. The individual used UDFs are listed in Table 5.1.

Table 5.1.: Used UDFs and their explanations

UDF	Explanation	Output	CFG-Blocks
late	TPC-H based, identify suppliers who have delivered orders late, scale was reduced to 0.065!	BOOLEAN	8
margin	TPC-H based, calculates the potential savings that can be achieved by replacing equivalent parts with cheaper parts.	FLOAT	15
packing	TPC-H based, divides the order items into different packages, taking into account the capacity limits	TEXT	17
ship	TPC-H based, finds the most used shipping method (air, ground, or mail) for a customer and returns it.	TEXT	5
supply	TPC-H based, savings that could be achieved by selecting the cheapest supplier for each element of an order.	FLOAT	10
mul	Multiplication by repeated addition until given number n.	TABLE	3
range	Generates an ascending sequence of numbers, starting from n to u.	TABLE	4
visible	calculates 3D landscape visibility by analyzing sightlines and terrain obstacles.	BOOLEAN	8

5.1. WITH [RECURSIVE] vs. umbra.trampoline

In order to measure the performance differences in detail, we ran each UDF with the Flummi compiler, whereby the specific flags for the DBMS were set to **Umbra**. Specifically, we compiled once with the `UMBRA_TRAMPOLINE` flag and once without this flag. This allowed us to isolate and examine the effects of the respective compilation settings.

The resulting output was then matched to the TPC-H database. This was done by modifying the **FROM** clauses to achieve the desired data volumes and to ensure that the test conditions were consistent and representative. These adjustments were necessary to create a realistic and practical test environment that reflected the actual performance of the UDFs in use.

In order to obtain a meaningful data basis, we carried out the tests under different conditions.

- **Number of used threads**

To measure the impact of active threads on execution time, we executed each UDF with 2, 4, 6, 8, 10 and 12 active threads.

- **Repetition**

To ensure the most meaningful results, the UDF was executed ten times in each thread constellation, and the median execution time was calculated. This method is more resistant to outliers [19], thus providing a more accurate representation of the data.

To enable a meaningful comparison, we used the best median results from the above-mentioned test conditions as a basis. This means that the thread constellation in which the respective version achieved the highest speed was selected. The results are listed in [Table 5.2](#): It should be noted that the **Performance Difference** column in [Table 5.2](#) must be interpreted as meaning that the `umbra.trampoline` variant works slower or faster than the `WITH [RECURSIVE]` variant by the specified percentage.

The **Speedup Ratio** column shows the factor by which the speed of the `WITH [RECURSIVE]` variant changes compared to the `umbra.trampoline` variant.

Table 5.2.: Comparison of the best median results under different thread constellations between **WITH [RECURSIVE]** and **umbra.trampoline**

UDF	WITH [RECURSIVE] (s)	trampoline (s)	Performance Difference	Speedup Ratio
TPC-H based udfs				
Ship	0.157	0.251	+60.41%	(1.60x)
Supply	1.949	3.372	+72.98%	(1.73x)
Margin	8.465	11.587	+36.91%	(1.37x)
Late	13.255	9.508	-28.29%	(0.72x)
Packing	25.079	44.688	+78.19%	(1.78x)
Non-device-based udfs				
mul	11.084	4.868	-56.10%	(0.44x)
range	22.672	13.258	-41.55%	(0.58x)
visible	4.168	4.016	-3.65%	(0.96x)

To gain a deeper understanding of the runtimes and to analyze the effects of different numbers of threads on performance, the median runtimes for each number of threads are graphically displayed below. These visualizations facilitate comparison of the performance of the different variants under varying conditions. [Figure 5.1](#) to [Figure 5.4](#) illustrate the UDFs **ship**, **supply**, **margin** and **packing** where the **WITH [RECURSIVE]** variant demonstrated superior performance compared to the **umbra.trampoline** variant.

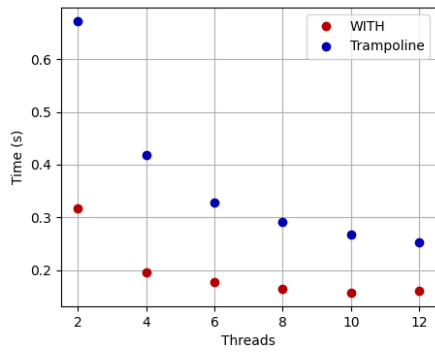


Figure 5.1.: Performance comparison of UDF ship

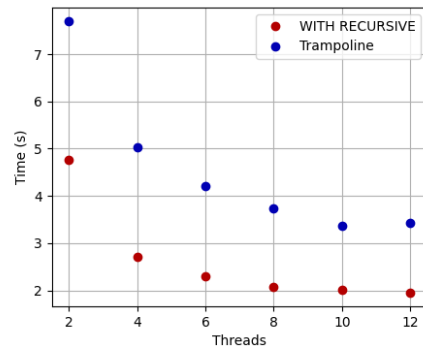


Figure 5.2.: Performance comparison of UDF supply

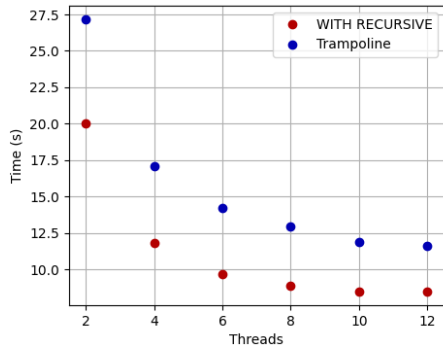


Figure 5.3.: Performance comparison of UDF margin

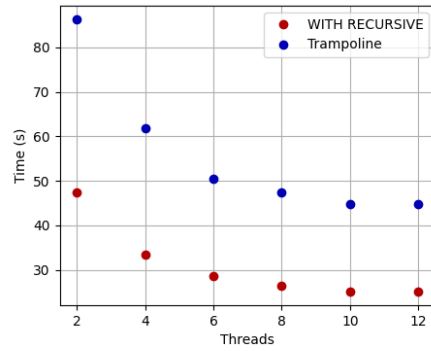


Figure 5.4.: Performance comparison of UDF packing

In contrast, [Figure 5.5](#) to [Figure 5.8](#) illustrate the UDFs `late`, `mul`, `range` and `visible` in which the `umbra.trampoline` variant demonstrates its capabilities and performs more effectively. These diagrams illustrate the scenarios in which the `umbra.trampoline` variant is superior to the `WITH [RECURSIVE]` variant.

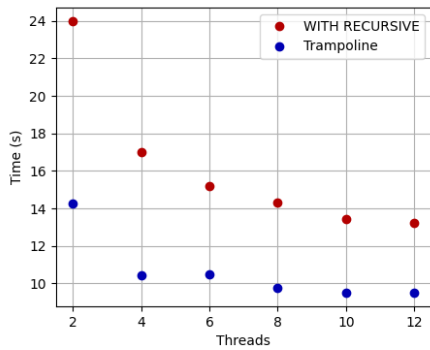


Figure 5.5.: Performance comparison of UDF late

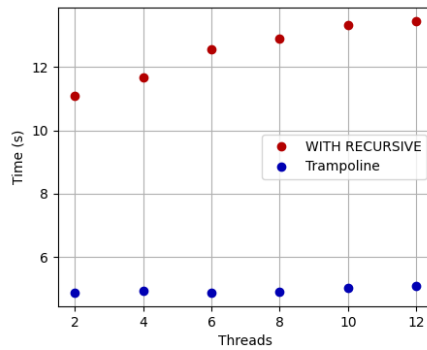


Figure 5.6.: Performance comparison of UDF mul

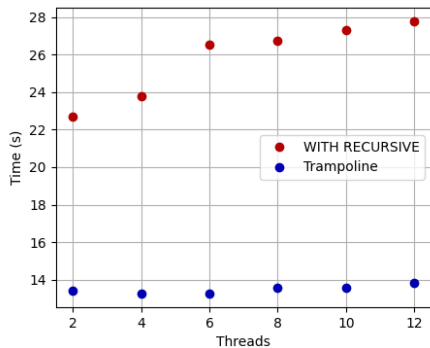


Figure 5.7.: Performance comparison of UDF range

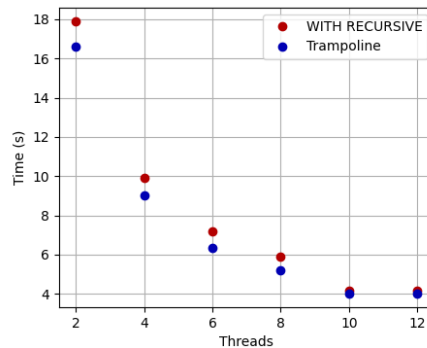


Figure 5.8.: Performance comparison of UDF visible

5.2. Umbra vs. DuckDB

To demonstrate the performance of the DBMS Umbra, we also executed all UDFs in DuckDB and recorded and compared their execution times. This was done on the same system used for the `WITH [RECURSIVE]` and `umbra.trampoline` comparisons. Since the trampoline feature is not available in DuckDB, we performed all comparisons on identical `WITH [RECURSIVE]` codes. Although the Flummi compiler provides additional `DuckDB` features that could improve performance, it was used here to ensure comparability.

Table 5.3 presents a comprehensive comparison of all determined times. To ensure the robustness and meaningfulness of the comparison, each UDF was executed ten times. The median value was calculated from these repetitions and used for the comparative analysis.

Table 5.3.: Equalization times between DuckDB and Umbra

UDF	DuckDB(in sec)	Umbra(in sec)	performance difference	speedup ratio
TPC-H based udfs				
Ship	0.917	0.160	+473.13%	(5.73x)
Supply	8.596	1.949	+341.05%	(4.41x)
Margin	20.894	8.465	+146.83%	(2.47x)
Late	29.686	13.255	+123.96%	(2.24x)
Packing	73.807	25.079	+194.30%	(2.94x)
Non-device-based udfs				
mul	13.029	0.141	+9140.43%	(92.41x)
range	25.001	0.277	+8925.63%	(90.25x)
visible	88.564	4.168	+2024.70%	(21.25x)

The data presented in the table provides a clear and unambiguous representation of the observed trends. The results demonstrate that **DuckDB** showed a notable decline in performance compared to **Umbra**. **Umbra** exhibited a distinct advantage in performance across all UDFs. A direct comparison with the original values was not feasible for the UDFs **mul** and **range**, as the execution times of **DuckDB** were excessively high. Consequently, the scope of the calculations in these two UDFs was significantly reduced, with this reduction applied equally to both DBMS. [Figure 5.9](#) demonstrates the notable differences in performance. It illustrates the superiority of **Umbra** over **DuckDB** with immediate visual clarity.

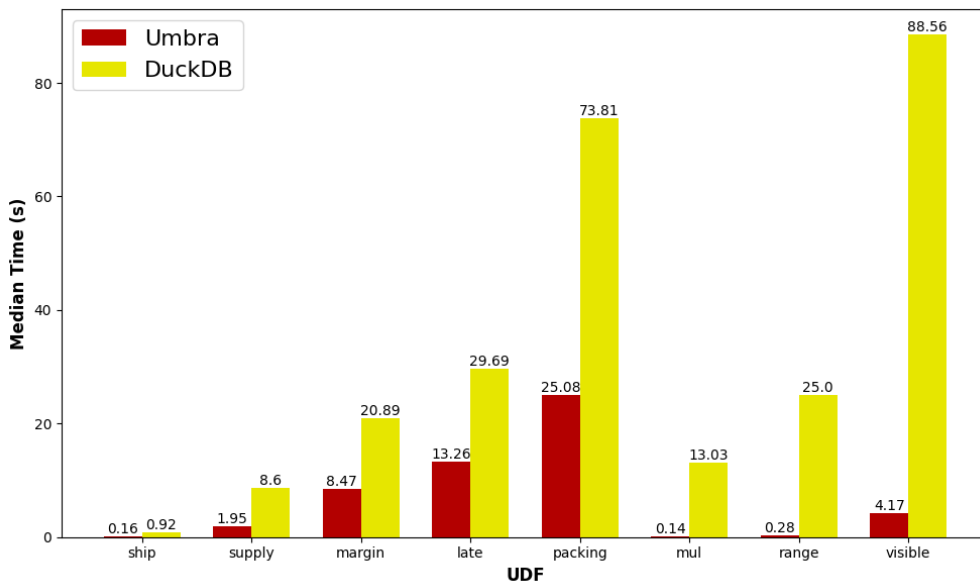


Figure 5.9.: Comparison of median times across different UDFs with identical WITH [RECURSIVE] variants

5.3. Discussion

A comparison with **DuckDB** reveals that **Umbra** is an highly potent DBMS. By combining in-memory technology with conventional SSD, **Umbra** is able to perform rapid evaluations even with large amounts of data. In the test scenario, **Umbra** demonstrated superior performance compared to **DuckDB**. It is anticipated that **Umbra** will further extend this lead as performance continues to scale. **DuckDB's** advantage of working efficiently in smaller environments and embedded systems is almost completely equalized if sufficient system performance is available.

In contrast, **Umbra** is capable of attaining its full performance potential due to its innovative buffer management and "versioned latches" technology, particularly in larger systems with a high number of threads, as is the case with server-based hardware. In the aforementioned example, the recursive UDFs `mu1` and `range`, which are seemingly simple, are especially conspicuous due to the performance deficiencies observed in **DuckDB**. This may be attributed to the fact that **DuckDB** does not employ just-in-time compilation, but instead relies on its proprietary query engine. Furthermore, in contrast to **Umbra**, **DuckDB** employs a less effective paging procedure, which may result in performance degradation when processing deep and complex queries. In conclusion, **Umbra** is designed from the outset for robust parallelization, which is less pronounced in **DuckDB**. Overall, **Umbra** is a highly capable system that exhibits commendable response times. Consequently, a comparison with a DBMS that shares similar objectives and requirements would have been a more appropriate choice in this context.

In comparing the `WITH [RECURSIVE]` technique with `umbra.trampoline`, however, the distinction between the two is less evident. The potential for a performance gain hinges on the specific queries to be processed. As previously stated, **Umbra's** aim is to achieve robust parallelization of evaluation processes. Consequently, trampolining, which also facilitates greater parallelization of individual task sections, should theoretically offer a speed advantage over the conventional `WITH [RECURSIVE]` evaluation. However, this is not the case for all UDFs. TPC-Hs-based ones in particular demonstrate that trampolining is not always the optimal solution. There may be several reasons for this. Firstly, the implementation in **Umbra** may not yet be sufficiently optimized, resulting in the lack of full exploitation of parallelization. Conversely, the optimizer could still exhibit clear advantages over the trampolining variant when evaluating `WITH [RECURSIVE]` queries. Furthermore, the small number of threads could also have led to the fact that the expected performance increases have not yet materialized.

As anticipated, the UDF `ship`, the only one without any loop, was incapable forging an advantage through trampolining. The `packing` and `supply` UDFs showed the most significant decline in performance due to trampolining, which may be attributed to the intricate nature of the CFG in these two queries. The UDFs `late`, `mu1`, and `range`, which have relatively few blocks in the CFG, demonstrated the most optimal performance in the comparison. This leads to the conclusion that the complexity of the UDF correlates with the performance, which could be improved by optimizing the feature in the implementation and the optimizer. However, further testing would be required to verify this hypothesis. Due to the limitations imposed on **Umbra** by missing features such as row types and other constructs, a comparison at a later stage with a more mature implementation would be more meaningful and could provide more accurate results.

In summary, it can be stated that the benefits of `umbra.trampoline` depend heavily on which inputs are to be processed. More complex evaluations with large and elaborate CFGs still seem to have an advantage in the original `WITH [RECURSIVE]` variant.

Conclusion & Future Work

This thesis addresses the question of how the compiler “Flummi” can translate iterative PL/Flummi code into raw SQL queries. The compiler employs CFGs to represent the flow of an imperative program. The compiler then optimizes the CFG by removing superfluous blocks and combining statements.

The rationale motivating this approach is that data can be maintained within the SQL environment, which offers a performance advantage when evaluating queries, as no prior data transfer is necessary. A primary objective was to eliminate the conventional CTE constructs of the compiler and instead utilize the distinctive `umbra.trampoline` functionality of the DBMS **Umbra**.

The `umbra.trampoline` feature is based on the method of “trapolining,” which involves restructuring iterative and recursive calculations into discrete steps, controlled by the trampoline operator. This restructuring allows for the avoidance of deep recursion and offers advantages when used in parallelized systems and multithreaded environments.

This function has been integrated into the compiler so that it can be called separately by specifying special flags to generate the desired trampoline outputs. Each block of the CFG has been converted then into a trampoline table. The result not only provides the correct syntax, but also improves readability and comprehensibility compared to the classic CTE constructs.

To evaluate the performance of the trampoline output, several UDFs were used, with some of the TPC-H benchmark database serving as a reference. The results showed that the expected performance improvements were observed in half of the cases. However, in some cases a significant speedup was achieved; for example, the UDF `mul` was 2.27 times faster than the variant with CTE.

Of the **eight** UDFs examined, **four** were able to achieve a performance increase through the trampoline functionality, while the remaining four showed a performance degradation in certain scenarios. It was found that the trampoline functionality gains greater benefits compared to the classic CTE variant as the number of threads increases.

The large differences in performance could be due to the not yet fully optimized functionality of the feature. However, it appears that the more complex the resulting CFG of the input program is, the worse `umbra.trampoline` performs compared to the classic `WITH`-variant. Further tests would be required to investigate whether certain control flow structures have a greater influence than others or whether a generally larger CFG leads to poorer results. It is also necessary to fully implement the feature in order to be able to make well-founded statements.

To illustrate the performance of **Umbra**, a detailed comparison with **DuckDB** was carried out. The results of this comparison clearly show that **Umbra** has a significantly higher performance

under the given test conditions. This comparison highlights **Umbra's** strengths and illustrates its superior efficiency in the scenarios tested. It should be noted that only a few aspects of the DBMS were examined as examples, which limits the significance of the results for other use cases.

In conclusion, the present work does not allow for a definitive conclusion to be drawn regarding the extent to which the trampoline feature offers the predicted performance advantages over classic CTE constructs, particularly the **WITH RECURSIVE** constructs. The analysis demonstrated that the anticipated enhancements were realized in half of the cases, underscoring the necessity for further inquiry and refinement. To reach a well-founded conclusion, additional tests and a more comprehensive investigation of the specific use cases are essential to ascertain the precise strengths and limitations of the trampoline feature in comparison to traditional CTE methodologies. However, if a large number of threads are available and the UDF possesses appropriate complexity, **umbra.trampoline** is indeed capable of achieving a significant performance gain.



Appendix: UDF MUL

Flummi output of UDF mul as `umbra.trampoline` version:

```
1 SELECT t.a, t.b, t."%result%"
2 FROM umbra.trampoline(
3 --Label:Initialize Table number: 0
4 TABLE(SELECT 1,
5 CAST((4) AS int) AS "a",
6 CAST((5000000) AS int) AS "b",
7 CAST(NULL AS int) AS "c",
8 CAST(NULL AS int) AS "%result%"
9 ),--Label:entry Table number: 1
10 TABLE(SELECT 3,
11 CAST(("a") AS int) AS "a",
12 CAST(("b") AS int) AS "b",
13 CAST((0) AS int) AS "c",
14 CAST(("result")) AS int) AS "%result%"
15 FROM trampoline
16 ),--Label:inter1 Table number: 2
17 TABLE(SELECT 0,"a", "b", "c",
18 CAST(("c") AS int) AS "%result%"
19 FROM trampoline
20 UNION ALL
21 SELECT CASE WHEN (TRUE AND NOT (0 = ("b"))) THEN 3
22 ELSE 0 END,
23 CAST(("a") AS int) AS "a",
24 CAST(("b") AS int) AS "b",
25 CAST(("c") AS int) AS "c",
26 CAST(("result")) AS int) AS "%result%"
27 FROM trampoline
28 WHERE (TRUE AND NOT (0 = ("b")))
29 ),--Label:multiplication_loop_head Table number: 3
30 TABLE(SELECT 2,
31 CAST(("a") AS int) AS "a",
32 CAST(("b") - 1) AS int) AS "b",
33 CAST(("a" + ("c")) AS int) AS "c",
34 CAST(("result")) AS int) AS "%result%"
35 FROM trampoline
36 )
37 AS t;
```

Listing A.1: `umbra.trampoline` mul version

Bibliography

- [1] T. Neumann and M. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *Conference on Innovative Data Systems Research (CIDR)* (2020).
- [2] A. Kemper and A. Eickler. *Datenbanksysteme*. 10. Auflage. De Gruyter Oldenbourg, 2015. ISBN: 978-3-11-044375-2.
- [3] S. J. Finkelstein, N. Mattos, I. Mumick and H. Pirahesh. “Expressing Recursive Queries in SQL”. In: *International Organization for Standardization* (1996).
- [4] C. Duta. “Another Way to Implement Complex Computations: Functional-Style SQL UDF”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA 2022), collocated with SIGMOD 2022. Philadelphia, PA, USA* (2022).
- [5] S. E. Ganz, D. P. Friedman and M. Wand. “Trampoline Style”. In: *ICFP99: International Conference on Functional Programming* (1999).
- [6] L. Lambrecht, A. Birler, T. Neumann and T. Grust. “Trampoline-Style Queries for SQL”. In: *under submission* (2024).
- [7] D. Hirn. “New Compilation Methods for Complex User-Defined Functions”. Doctoral Dissertation. PhD thesis. Eberhard Karls Universität Tübingen, 2024.
- [8] statista. “The most popular programming languages worldwide”. In: (2024). URL: <https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebtteste-programmiersprachen-weltweit-laut-pypl-index/>.
- [9] J. Ernesti and P. Kaiser. *Python 3, Das umfassende Handbuch*. 6. Auflage. Rheinwerk Verlag, Bonn 2021, 2021. ISBN: 978-3-8362-7926-0.
- [10] T. Fischer, D. Hirn and T. Grust. “SQL Engines Excel at the Execution of Imperative Programs”. In: *VLDB 2025, 51th International Conference on Very Large Data Bases (VLDB 2025), London* (2025).
- [11] Technische Universität München Institut für Informatik Lehrstuhl III: Datenbanksysteme (I3). *The Umbra Database System*. visited on 10 June 2024. 2024. URL: <https://umbra-db.com/>.
- [12] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.18: Conditional Expressions*. visited on 13 August 2024. 2024. URL: <https://www.postgresql.org/docs/current/functions-conditional.html#FUNCTIONS-COALESCE-NVL-IFNULL>.
- [13] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 1: SQL Syntax*. visited on 12 August 2024. 2024. URL: <https://www.postgresql.org/docs/7.2/sql-syntax.html>.
- [14] W3Schools. *W3Schools Online Web Tutorials: Python String join() Method*. visited on 12 August 2024. 2024. URL: https://www.w3schools.com/python/ref_string_join.asp.

- [15] M. Raasveldt, P. Holanda, T. Gubner and H. Mühleisen. "Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing". In: *DBTest '18: Proceedings of the Workshop on Testing Database Systems* (2018).
- [16] DuckDB-Team. *DuckDB: An Embeddable Analytical Database*. visited on 24 July 2024. 2024. URL: <https://duckdb.org/>.
- [17] DuckDB-Team. *Performance Environment Guide*. visited on 24 July 2024. 2024. URL: <https://duckdb.org/docs/guides/performance/environment>.
- [18] Transaction Processing Performance Council. *TPC Benchmark™ H (TPC-H)*. visited on 17 July 2024. 2024. URL: <https://www.tpc.org/tpch/>.
- [19] G. Teschl and S. Teschl. *Mathematik für Informatiker: Band 2: Analysis und Statistik*. 3. Auflage. Springer Vieweg Berlin, Heidelberg, 2015. ISBN: 978-3-642-54273-2.