Partial UDF Inlining

Samuel Arch[1], Yuchen Liu[1], Todd C. Mowry[1], Jignesh M. Patel[1], Andrew Pavlo[1]
[1]Carnegie Mellon University
sarch@cs.cmu.edu, yuchenl6@andrew.cmu.edu
tcm@cs.cmu.edu, jignesh@cmu.edu, pavlo@cs.cmu.edu

ABSTRACT

Although user-defined functions (UDFs) are a popular way to augment SQL's declarative approach with procedural code, the mismatch between programming paradigms creates a fundamental optimization challenge. UDF inlining automatically removes all UDF calls by replacing them with equivalent SQL subqueries. Although inlining leaves queries entirely in SQL (resulting in large performance gains), we observe that inlining the *entire* UDF often leads to suboptimal performance. A better approach is to analyze the UDF, deconstruct it into smaller pieces, and inline only the pieces that help query optimization. To achieve this, we propose UDF outlining, a technique to intentionally hide pieces of a UDF from the optimizer, resulting in simpler UDFs and significantly faster query plans. Our implementation (PRISM) demonstrates that UDF outlining improves performance over conventional inlining (on average 1.29× speedup for DuckDB and 298.73× for SQL Server) through a combination of more effective unnesting, improved data skipping, and by avoiding unnecessary joins.

1. INTRODUCTION

Application developers extend SQL's capabilities by incorporating user-defined functions (UDFs) written in other programming languages (e.g., Python, JavaScript, PL/SQL) into their queries. However, queries with UDF calls are challenging for a database system's query optimizer to reason about when choosing a good query plan because they are opaque functions written in a non-relational paradigm. As a result, queries with UDFs are often orders of magnitude slower than equivalent queries written without UDFs.

To address this impedance mismatch, researchers developed optimization techniques for UDFs, including *compilation* (i.e., generating specialized machine code for the UDF) [39, 37], *batching* (i.e., coalescing individual UDF invocations) [20, 15], and more recently, *UDF inlining* (i.e., translating a UDF into an equivalent SQL subquery) [45, 42, 22]. Inlining has shown the most potential of these techniques, providing up to 1000× performance improvements for workloads in a commercial DBMS [5].

The effectiveness of inlining stems from its ability to represent UDFs as SQL subqueries, a relational form that the DBMS can optimize. For example, `x=y` becomes `SELECT y AS x`, and `IF/ELSE` blocks become `CASE WHEN` expressions. The DBMS then chains these translated expressions together using `LATERAL` joins, resulting in a SQL subquery equivalent to the original UDF. Inlining then replaces the original UDF call with the generated subquery that it injects into the calling query, leaving it in a purely relational form that enables the query optimizer to find better query plans.

The problem, however, is that inlining leaves many UDFs in an obfuscated form that the DBMS cannot reason about and optimize effectively. Figure 1 shows a UDF-centric query from Microsoft's SQL ProcBench [19] (a benchmark modeled after Azure customer workloads). The figure demonstrates that inlining the *entirety* of the UDF generates complex subqueries containing `LATERAL` joins that are (1) challenging to unnest and optimize and (2) slow to execute.

A better approach is to deconstruct the UDF, identify the fragments beneficial for inlining, and expose them to the query optimizer. Now, UDF-centric queries become simpler and `LATERAL`-free by inlining only the code fragments necessary for better query plans. However, deconstructing a UDF is challenging for several reasons. First, users mix procedural and relational code by embedding `SELECT` statements inside control flow (conditionals/loops), preventing the DBMS from applying different optimization techniques to the same code block. Second, deciding which code to inline while minimizing code size is difficult. Lastly, users often write UDF predicates in `WHERE` clauses, which existing optimization techniques fail to exploit for data-skipping.

Given this, we present the **PRISM** UDF optimization framework. When an application registers a new UDF (i.e., with a `CREATE FUNCTION` command), PRISM performs analysis to carefully deconstruct the function into pieces, inlining some pieces and intentionally hiding others from the optimizer. The critical technique PRISM uses to achieve this is UDF *outlining*, which extracts UDF code fragments into separate functions that are intentionally not inlined, minimizing UDF code complexity. With the ability to either inline or outline pieces of a UDF, PRISM strategically restructures UDFs, maximizing the outlined pieces to reduce UDF complexity while inlining any pieces that may lead to better query plans. In addition, PRISM injects UDF predicates into a query's `WHERE` clause to make data-skipping opportunities transparent to the optimizer.

To achieve this, PRISM performs five complementary UDF-centric optimizations that combine to address the drawbacks of inlining: (1) Predicate Hoisting, (2) Region-Based UDF

© Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. This is a minor revision of the paper "The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining", published in PVLDB, Vol. 18, No. 1, ISSN 2150-8097. DOI: https://doi.org/10.14778/3696435.3696436

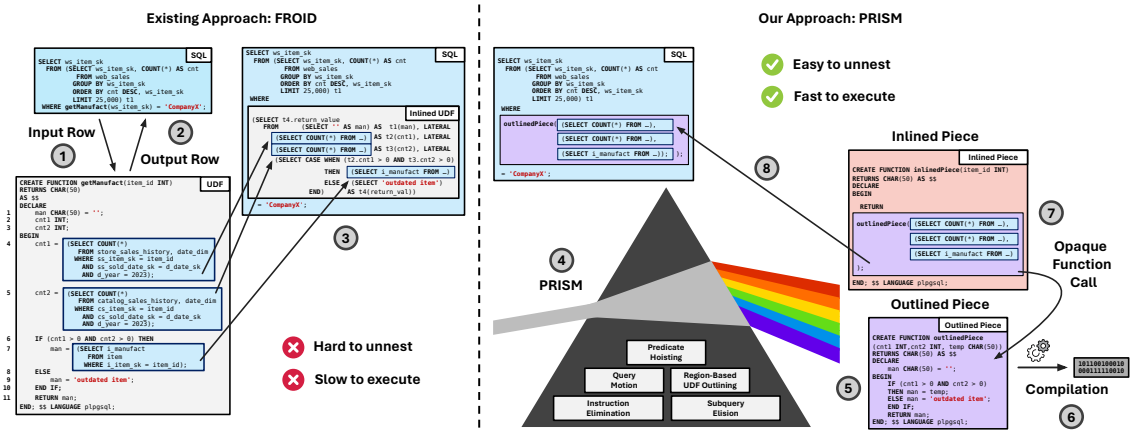


Figure 1: Overview of PRISM– The existing approach to UDF optimization (FROID) translates UDF execution from row-by-row oriented execution into a SQL subquery that it inlines into into the calling query. Our approach (PRISM) deconstructs a UDF into separate inlined and outlined UDF pieces. The framework injects inlined pieces into the calling query, but then compiles outlined pieces to machine code so that they are opaque to the DBMS's optimizer.

Outlining, (3) Instruction Elimination, (4) Subquery Elision, and (5) Query Motion).

We integrated PRISM into DuckDB [41] and Microsoft SQL Server to evaluate our approach. Our experimental results show that after PRISM simplifies UDFs, they become straightforward for DBMSs to optimize and execute. As a result, PRISM accelerates UDF-centric queries from the Microsoft ProcBench [19] (by an average of 1.3× on DuckDB and 298.7× on SQL Server, and by a maximum speedup of 2270.2× and 2997.9×, respectively).

We make the following contributions in this paper:

1. We identify the performance challenges introduced by UDF inlining during query optimization and execution: complex sub-queries containing `LATERAL` joins generated when inlining the entirety of a UDF (Section 2.4).
2. We introduce *UDF outlining*, a method to extract pieces of a UDF into separate functions to avoid inlining them, enabling more effective UDF optimization (Section 3.1).
3. We propose four complementary UDF-centric optimizations that restructure a UDF (Section 4), inlining only the pieces that lead to fast plans while maximizing the amount of outlined code and hoisting UDF predicates for data-skipping.

2. BACKGROUND

UDFs allow users to extend the database systems' functionality by mixing imperative and declarative programming paradigms. But they create an impedance mismatch between UDFs and SQL that is challenging for both query optimization and execution. As we now describe, these problems have led to the development of methods to automatically optimize UDFs, but they all have drawbacks.

2.1 Compilation

Some DBMSs employ *compilation* techniques to reduce the engineering overheads associated with UDFs. As early as 2001, Oracle added support for "native compilation" of PL/SQL UDFs [39], transpiling them to C code, compiling the code into a shared library. This allows the DBMS to execute a UDF as though it were a built-in function. Oracle also provides the `PRAGMA UDF` knob to allow the DBMS to reuse memory frames/data structures for function argument passing to reduce context switching overhead [12] .

Microsoft SQL Server similarly added the ability to compile T-SQL UDFs to machine code in 2016 [37]. SingleStore added support to compile PL/pgSQL UDFs to LLVM IR in 2021 [46]. For dynamically-typed UDFs (i.e., Python UDFs), just-in-time (JIT) compilation is essential and is adopted by systems such as YeSQL [14, 13], BabelFish [17], Tuplex [48, 47], and Actian Vector [30]. However, compilation is ineffective if the chosen plan is sub-optimal.

2.2 Batching

Another earlier optimization technique is to *batch* invocations to amortize context-switching overheads of executing UDFs one row at a time. In 2008, researchers first proposed batching UDF invocations using program rewriting rules [20]. This approach only applies to DBMSs with a UDF interpreter rather than translating the batched queries to pure SQL. Later work refined UDF batching by rewriting UDF-centric queries to execute as a sequence of `UPDATE` statements on a temporary table [15]. Although this technique simplifies UDF-centric queries substantially, the overhead of materializing temporary tables is prohibitive.

2.3 Inlining

For a DBMS to achieve truly excellent performance for UDF-centric queries, the optimizer must reason about the UDF's semantics as if it were written in SQL. UDF inlining [45] (developed in 2014) accomplishes this by translating UDFs into relational algebra (RA) that the DBMS can optimize effectively. Inlining translates each procedural instruction in a loop-free PL/SQL UDF to an equivalent RA expression. `IF/ELSE` blocks become `CASE WHEN` statements, assignments (i.e., `x = y`) become projections (i.e., `SELECT y AS x`). Then, the DBMS chains together the translated statements with `APPLY` operators [1], creating a single RA expression which is equivalent to the original UDF. Although the

[1]T-SQL's `APPLY` is similar to the SQL:1999's `LATERAL` join.

2014 approach laid the foundation for future research, the main drawback is that it requires modifying the query optimizer to support extensions of the `APPLY` operator, which is not possible in closed-source systems and requires significant testing to avoid regressions.

Microsoft's **FROID** pioneered a translation strategy to convert UDFs to relational algebra without requiring extensions to `APPLY` [42]. After inlining a UDF, the DBMS employs parallel, set-oriented execution plans instead of inefficient, serial, iterative plans. Microsoft released FROID with SQL Server 2019 and showed up to 1000× performance improvements for customer workloads [5].

Although FROID only supports inlining of loop-free UDFs, further techniques have since lifted this restriction. Aggify [18] for example, uses dataflow analysis to replace all cursor loops inside a UDF with custom aggregate functions, enabling the UDF to then be inlined. However, both FROID and Aggify require changes to the DBMS internals.

More recently, the **Apfel** framework provides a pure SQL translation for UDFs [22]. For a given UDF, Apfel converts its procedural logic into a series of `SELECT` statements representing the same computation. The framework combines these statements into a single `SELECT` statement using `LATERAL` joins and then inlines the statement into the calling query. Apfel supports arbitrary control flow (including loops) via recursive common table expressions (CTEs). Any DBMS that supports `LATERAL` joins can execute inlined UDFs with Apfel, including systems without native UDF support.

2.4 Motivation

Inlining translates UDFs to complex subqueries containing `LATERAL` joins that are challenging for the DBMS to optimize and execute efficiently for the following three reasons:

(1) Difficult to Unnest: Inlining generates complex subqueries that are challenging for the DBMS to unnest (i.e., replace with join operators). Prior work shows that inlining UDFs produces subqueries that widely used DBMSs cannot unnest [15]. As a result, after inlining, the DBMS invokes the subquery once for each input row (similarly to how the original query invoked the UDF), resulting in extremely slow query execution. Although newer DBMSs like DuckDB unnest arbitrary queries [41, 38], existing systems do not perform arbitrary unnesting as it requires invasive changes to the query optimizer (using DAG-shaped rather than tree-shaped query plans, introducing special join operators). As we will show in Section 5, even for DuckDB, which supports arbitrary unnesting, these complex subqueries introduce additional join operators during the unnesting process, slowing down query performance.

(2) Prevent Data Skipping: Users often invoke UDFs as predicates in a query's `WHERE` clause (see Figure 1). With inlining, however, the DBMS replaces UDFs with subqueries that it cannot push down into the leaves of the query plan. These scenarios prevent the DBMS from employing data-skipping optimizations (i.e., block skipping using zone maps or index scans), which causes queries to unnecessarily scan an entire table sequentially. Inlining obfuscates predicates, thereby blocking the optimizer from reasoning about them and identifying opportunities to avoid unnecessary work.

(3) Inefficient Query Execution: By translating a UDF into a sequence of `LATERAL` joins, inlining creates inefficient queries because DBMS optimizers struggle with join operators. Furthermore, inlining embeds these `LATERAL` joins inside a subquery, which, after unnesting, the DBMS replaces with a join, incurring additional overhead. Lastly, as we will show in Section 5, inlining translates loops into Recursive Common Table Expressions (CTEs) that often are an order of magnitude slower to execute than the loop in procedural form. Such issues argue for a UDF optimization technique that reduces the number of joins in a query.

3. PRISM OVERVIEW

PRISM is a UDF optimization framework that deconstructs a UDF into separate inlinable and outlinable pieces. Its goal is to inline the UDF pieces, exposing the code most amenable to SQL-style execution to the query optimizer while outlining as much of the remaining code as possible.

As shown in Figure 1, when the application installs a new UDF into the database (i.e, `CREATE FUNCTION`), PRISM examines the function's contents to identify (1) which parts to inline into the calling query and (2) which parts to compile into machine code as separate functions through outlining. By only inlining a small portion instead of the entire UDF, PRISM eliminates all `LATERAL` joins. Thus, outlining provides two benefits: (1) it hides the outlined code from the optimizer through an opaque function, thereby minimizing the UDF complexity and resulting in simpler queries for the DBMS to optimize, and (2) the system can compile the outlined code to machine code to improve performance. As a result, PRISM's optimizations overcome all the limitations described in Section 2.4.

We now describe PRISM's architecture in more detail. We then discuss PRISM's intermediate representation of UDFs to support our new optimizations in Sections 3.2 and 3.3.

3.1 Architecture

Figure 1 depicts the overall architecture of PRISM for the motivating example from Section 1. Compared to existing approaches which operate at query time (replacing the row-by-row UDF execution ① ② by inlining the UDF into the calling query ③), our approach (PRISM) occurs when a `CREATE FUNCTION` command is registered with the system. At this stage PRISM parses, binds, and translates the UDF to an intermediate representation (IR) (shown in Figure 2). PRISM then applies novel compiler optimizations ④ .

First, PRISM decouples procedural and relational code with query motion (Section 4.1) to expose the largest possible code sequences for outlining. Next, UDF outlining (Section 4.2) extracts these code sequences into separate outlined pieces ⑤, which are then compiled to machine code ⑥, replacing the original code with opaque function calls ⑦ that minimize the UDF complexity. Then, instruction elimination (Section 4.3) removes as many instructions in the UDF as possible, collapsing a UDF down to a single `RETURN` instruction that does not require `LATERAL` joins when inlined ⑦. Subquery elision (Section 4.4) then replaces the original UDF call with the return value of the UDF ⑧, bypassing the inlining step and avoiding the generated subquery. Lastly, predicate hoisting analyzes loop-free UDFs and saves them as boolean predicates injected into a query's `WHERE` clause, making data-skipping optimizations transparent to the optimizer. Unlike existing approaches that produce subqueries that are hard to unnest and slow to execute, PRISM produces `LATERAL`-free queries that are easy to unnest and faster.

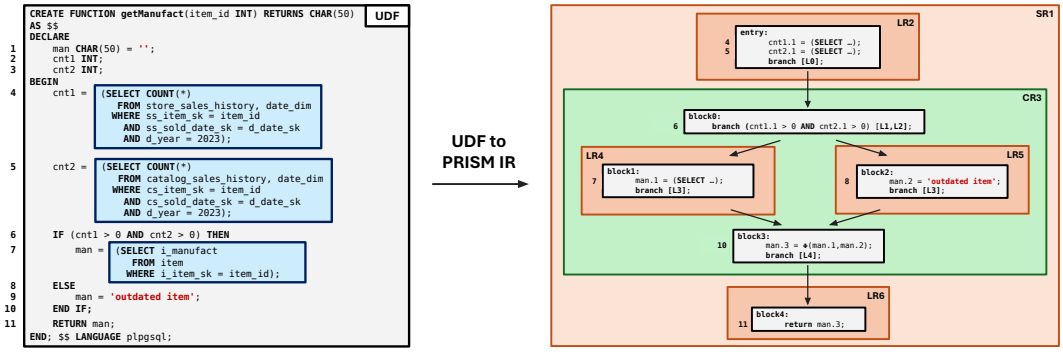


Figure 2: PRISM's Intermediate Representation (IR) – PRISM uses an SSA-based IR, ensuring variables are assigned exactly once in the entire program. The IR also encodes high-level structured control flow (conditionals, loops) as a hierarchy of program regions. **LR** denotes a leaf region, **CR** denotes a conditional region, and **SR** denotes a sequential region.

3.2 Static Single Assignment (SSA) Form

PRISM represents UDFs as *control flow graphs* (CFGs) in *static single assignment* (SSA) form [43, 4] (shown in Figure 2). The CFG simplifies program analysis by breaking down high-level procedural constructs (loops, conditionals) into conditional and unconditional jumps (branches) between basic blocks [2]. Each basic block holds a sequence of non-branching instructions (assignments, function calls), followed by a single terminator instruction (a jump or return) that ends the basic block. In the CFG, nodes represent basic blocks and jumps create a directed edge from the source to the target block.

SSA form is an intermediate representation (IR) that ensures each variable is assigned exactly once. SSA form simplifies compiler design by making data flow explicit in the IR and is used in nearly all modern compiler implementations (e.g., LLVM) [34, 31, 49].

To convert to SSA form, existing variables (i.e., *man* from Figure 2) are assigned versions (i.e., *man.1*, *man.2*, *man.3*), one for each program point where the variable is assigned a value. At join points in the CFG (**block3**), ϕ-functions are inserted (i.e., *man.3*=ϕ(*man.1, man.2*)) where each ϕ argument indicates the variable's value when control flow passes from a given predecessor. Note that ϕ functions are not executable, and the compiler eventually converts the program out of SSA form. Although PRISM uses SSA for simplicity, our optimizations also apply to non-SSA programs.

3.3 Regions

PRISM represents UDFs as a program structure tree [26], a hierarchy of single-entry single-exit (SESE) program *regions* [21, 1] (shown in Figure 2). Unlike the CFG that represents control flow as jumps, regions retain the high-level structured control flow of the original program (conditionals, loops) [26, 21, 1], simplifying certain compiler optimizations. There are four types of regions: leaf regions (a single basic block), conditional regions (blocks contained in an `IF/ELSE`), loop regions (blocks contained within loops), and sequential regions (a sequence of nested regions).

In Figure 2, the entire UDF is a single sequential region **SR1** comprised of nested subregions. The subregions are leaf region **LR2**, followed by conditional region **CR3**, and terminating with leaf region **LR6**. **CR3** contains block **block0**, which branches control flow between the `IF` and `ELSE` regions denoted by **LR4** and **LR5**, respectively, followed by the fall through block **block3**, which joins control flow from the previous regions.

During SSA construction, ϕ functions, conditional branch instructions, and `SELECT` statements are placed in individual basic blocks ensuring that PRISM can cleanly construct program regions. By placing `SELECT` statements in leaf regions, non-`SELECT` code can be outlined independently (described in Section 4.2). By representing the program as a hierarchy of regions, each region has a single entry and exit point, enabling PRISM to outline regions as separate functions (shown in Section 4.2).

4. UDF-CENTRIC OPTIMIZATIONS

PRISM contains a suite of UDF-centric optimizations that restructure a UDF by splitting it apart into pieces and then optimizing them through a combination of UDF outlining and inlining. These optimizations are complementary and are meant to be combined together to achieve the best performance for UDF-centric queries.

For the examples in this section, we illustrate PRISM's code changes to the UDF using (1) **blue** to represent SQL code (i.e., embedded `SELECT` statements, queries), (2) **red** to represent deleted code, and (3) **green** to represent new code.

4.1 Query Motion

Users often place `SELECT` statements inside of UDF control flow, such as conditionals and loops (including in our motivating example). However, mixing relational and procedural code introduces a challenge for UDF optimization. On the one hand, the DBMS should inline `SELECT` statements, making them transparent to the query optimizer. On the other hand, the DBMS should outline procedural control flow into opaque function calls, minimizing the UDF code size and the number of `LATERAL` joins that inlining generates. Coupling the relational code with procedural control flow prevents outlining of the procedural code, forcing the DBMS to inline the entire region and produce complex, hard-to-optimize subqueries.

A better approach is for the DBMS to *hoist* the `SELECT` statements outside of the procedural code (shown in Figure 3), splitting the UDF into pieces containing only procedural code (optimized for outlining) and relational code (optimized for inlining)[2]. At the same time, it must carefully

[2]Although PRISM performs its optimizations on its inter-

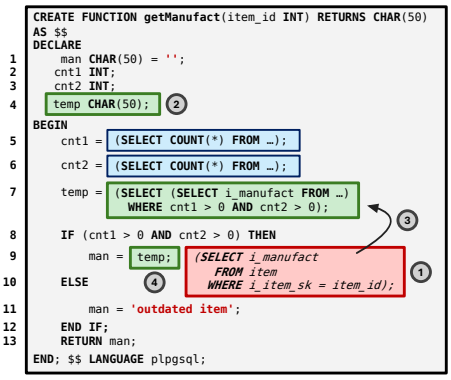


Figure 3: Query Motion – Hoisting `SELECT` statements outside of control flow (loops, conditionals) to expose larger code regions for outlining.

hoist the `SELECT` statement to avoid introducing redundant work and performance regressions.

Figure 3 illustrates the hoisting process for our motivating example. ① PRISM first chooses the `SELECT` statement on line 9 as a candidate for hoisting. Next, ② PRISM determines line 7 as the highest program point for hoisting and adds a temporary variable *temp* of the same type as the `SELECT` to the UDF. Then, ③ PRISM assigns the return value of the `SELECT` to the temporary, where the condition from line 8 now predicates the statement. Lastly, ④ PRISM rewrites the original assignment on line 9 to assign from the temporary instead of from the `SELECT` statement.

After query motion, PRISM decouples procedural and relational code, exposing larger regions of procedural code for UDF outlining. It is not always possible to hoist a `SELECT` statement outside of control flow (e.g., a loop that iteratively executes a `SELECT` that uses loop variables). In these cases, the DBMS falls back to inlining.

4.2 Region-Based UDF Outlining

With query motion, PRISM hoists `SELECT` statements above the procedural code, preparing the procedural code for UDF outlining. Although PRISM could inline the UDF at this stage, it is better to perform outlining for two reasons. First, outlining the procedural code into opaque function calls prevents the DBMS from inlining the functions, thereby reducing the number of `LATERAL` joins in the generated code and leading to more straightforward queries for the DBMS to optimize and execute. Second, after outlining, PRISM compiles the procedural code, making it an order of magnitude faster than if it were inlined. As a consequence, outlining as much procedural code as possible from the UDF is crucial to achieving high performance.

Outlining the maximum amount of procedural code into separate functions is non-trivial for several reasons. Query motion cannot hoist all `SELECT` statements, leaving some embedded inside procedural code blocks. Compiling these `SELECT` statements would hide them from the query optimizer, resulting in slow, iterative query plans [42], so PRISM leaves them for inlining. Yet, embedded `SELECT` statements (i.e., relational code in an `IF/ELSE` block) should only prevent outlining of the smallest surrounding code block (i.e., the `IF/ELSE`), not the other code blocks. Next, arbitrary UDF code may have multiple entry or exit points (multiple predecessor or successor blocks in the CFG), complicating code extraction. Lastly, PRISM should extract the largest possible procedural code at a time, minimizing the number of opaque function calls into and out of the UDF and exposing longer code sequences for compiler optimization.

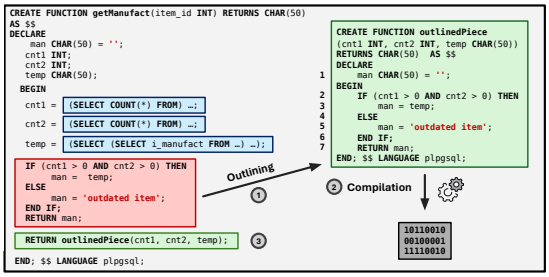


Figure 4: Region-Based UDF Outlining – Finding the UDF's largest procedural code regions and extracting them into functions for compilation.

The key insight necessary in addressing these challenges is to use a *region-based* approach to UDF outlining. As shown in Figure 2, PRISM represents UDFs as a program structure tree (a hierarchy of program regions) that represent structured control flow as a composition of four region types: (1) *sequential*, (2) *conditional*, (3) *loop*, and (4) *leaf*. Regions have precisely one entry and exit point [21, 1], ensuring that PRISM can extract any region into its own function. Further, PRISM handles embedded `SELECT` statements by never outlining a region if it contains another `SELECT` inside of it (including in its subregions).

Figure 4 illustrates the outlining process for our example UDF. First, ① PRISM identifies the largest region (lines 8–14) eligible for outlining. PRISM uses liveness analysis [1] to track variables entering the region (*cnt1*, *cnt2*, *temp*) and exiting the region (*man*). PRISM extracts this region into a separate function, with the entering variables becoming input arguments and the exiting variables becoming return values; PRISM creates user-defined types to handle multiple return values, similar to Aggify [18]. Next, ② the algorithm transpiles the extracted function to a C++ program, compiles it using `clang/gcc`, and then dynamically links it into the DBMS. Lastly, ③ the system removes the outlined region from the UDF and replaces it with a call to the compiled function. The extracted region becomes opaque to the query optimizer, resulting in queries that are simpler for the DBMS to optimize.

4.3 Instruction Elimination

Although region-based UDF outlining reduces UDF complexity, the resulting function still contains instructions that will produce a `LATERAL` join with inlining. Thus, PRISM must eliminate as many instructions as possible to produce simple, `LATERAL`-free queries that are straightforward to optimize. The challenge lies in how PRISM eliminates instructions while maintaining the UDF's correctness and not introducing performance regressions.

We refer to each line of code in the UDF as an *instruction* and embedded queries in the UDF as `SELECT` *statements*. An instruction is eligible for elimination if it is *dead* (i.e., an instruction that does not affect the program's result).

mediate representation, we show the transformations on PL/pgSQL UDFs in our figures.

Therefore, PRISM must ensure that no other instructions in the program depend on it by tracking whether an instruction updates the state of a variable that is not used by any subsequent instructions. PRISM accomplishes this by taking advantage of the key property of SSA form: that each variable is assigned exactly once in the entire program. Therefore, to eliminate an instruction (i.e., $y = f(x)$) from a UDF, PRISM replaces every use of a variable with its definition (i.e., y with $f(x)$), eliminating the variable (i.e., y) from the program and making the defining instruction redundant.

When a variable has multiple uses, it introduces an important optimization decision. PRISM could duplicate the expression in the program, removing the instruction and reducing the number of `LATERAL` joins. Yet duplicating the expression may result in the DBMS evaluating the expression multiple times and performing worse than the original UDF. On the other hand, by not duplicating the expression, the UDF will still contain instructions that will result in complex subqueries with `LATERAL` joins after inlining.

PRISM addresses this trade-off by considering the number of uses and the cost of evaluating the expression. If a variable has one use, PRISM replaces it with the definition, saving a `LATERAL` join in the inlined UDF without causing regressions. However, when a variable has multiple uses, PRISM will treat it differently depending on whether the expression is a `SELECT` statement. If the expression is not a `SELECT`, then PRISM always duplicates it, relying on the DBMS's common-subexpression elimination (CSE) pass (available in both DuckDB [7] and SQL Server [42]) to identify duplicated expressions and evaluate them only once.

For `SELECT` statements, PRISM chooses whether to duplicate the expression depending on the target DBMS that will execute the UDF. For DBMSs that unnest arbitrary queries [38] (e.g., DuckDB), PRISM does not perform query duplication. But for DBMSs that only unnest `LATERAL`-free subqueries (e.g., SQL Server), PRISM applies query duplication, minimizing the UDF complexity as much as possible to enable unnesting. Only three out of 29 UDF-centric queries in our experimental analysis in require PRISM to decide whether to duplicate `SELECT` statements.

If an instruction defined by a `SELECT` has multiple uses and the DBMS unnests arbitrary queries [38], then PRISM does not eliminate it. Otherwise, the algorithm replaces every use of the instruction with its definition and removes the defining instruction from the UDF. It skips instructions that reference themselves (i.e., a cyclic dependency).

Figure 5 illustrates the instruction elimination process for the motivating example. First, PRISM replaces the uses of ① *cnt1* on lines 6–7 with its definition and then ② does the same for *cnt2*. Next, ③ PRISM replaces *temp* on line 7 with its definition on line 6. The last step ④ removes all dead variables from the UDF.

Through instruction elimination, PRISM collapses the entire UDF into a single `RETURN` instruction, resulting in a `LATERAL`-free subquery after inlining. Since DuckDB unnests arbitrary queries, PRISM skips code duplication in step ① since *cnt1* and *cnt2* have multiple uses. After instruction elimination, PRISM removes as many of the instructions in the UDF as possible, leaving the UDF in a much simpler form (in almost all cases, as a single `RETURN` instruction), which is easier for the DBMS to optimize and execute.

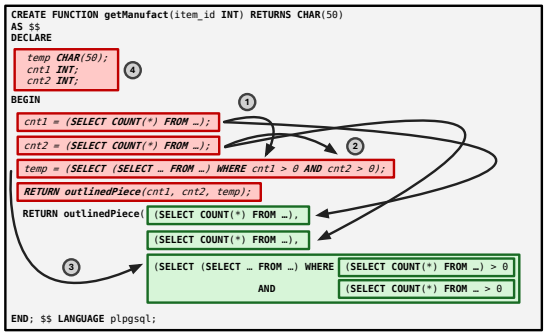


Figure 5: Instruction Elimination – Replacing each use of a variable with its definition, eliminating the instructions in a UDF, collapsing it down to a single `RETURN` instruction.

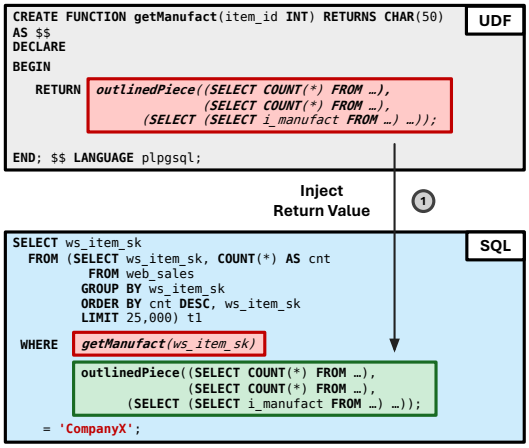


Figure 6: Subquery Elision – Replacing the UDF call with its return value, bypassing UDF inlining and the corresponding subquery.

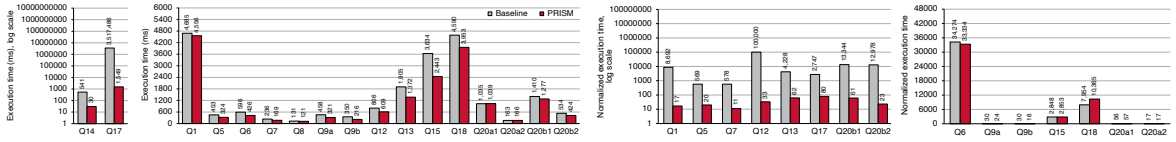
4.4 Subquery Elision

Whenever possible, PRISM collapses UDFs into a single `RETURN` instruction to ensure `LATERAL`-free queries after inlining. However, inlining still wraps UDFs in subqueries that complicate query optimization and execution. To overcome this, PRISM performs *subquery elision* to replace the original UDF call with its return value, skipping inlining and avoiding the introduction of an unnecessary subquery.

Figure 6 illustrates subquery elision for our motivating example. Instead of inlining the UDF and generating a subquery (i.e., `SELECT outlinedPiece( ...)`), ① the system directly substitutes the return value into the calling query. Although our experiments found that FROID already performs subquery elision whenever possible, to the best of our knowledge, we are the first to identify this optimization as a necessary step for effective UDF optimization.

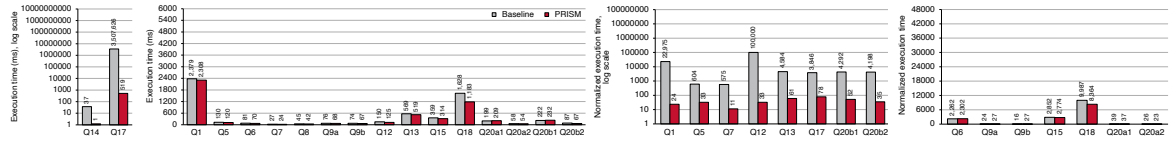
5. EXPERIMENTAL ANALYSIS

We now present an evaluation of PRISM in DuckDB [41] (commit `53dc13d` with a patch applied to support parallel CTEs [23]) and Microsoft SQL Server 2022. By integrating PRISM into DuckDB, the DBMS natively executes UDFs after registering them with the `CREATE FUNCTION` command.



(a) DuckDB ProcBench (Threads = 1) **(b)** SQL Server ProcBench (Threads = 1)

Figure 7: ProcBench (Single-Threaded) – Single-threaded execution times for DuckDB and SQL Server for the ProcBench queries with inlining ("Baseline") and PRISM. Execution times are normalized for SQL Server (setting the maximum value to 100,000 units). Queries executing more than 10× faster with PRISM are shown log-scale on the left-hand side, with the remaining queries shown on a linear scale.



(a) DuckDB ProcBench (Threads = 32) **(b)** SQL Server ProcBench (Threads = 80)

Figure 8: ProcBench (Multi-Threaded) – Multi-threaded execution times for DuckDB and SQL Server for the ProcBench queries with inlining ("Baseline") and PRISM.DuckDB was executed with 32 threads to avoid a known bug with non-terminating queries [6],and SQL Server was run with the maximum degree of parallelism available (`maxDOP`=80).SQL Server results are displayed normalized, and queries with a 10× speedup or greater are displayed log-scale.

To compare against inlining, we used Apfel [24] to generate the inlined UDF code for DuckDB. Since Apfel uses a different strategy than FROID, sometimes it generates slower queries than FROID. In these cases, we manually rewrite the queries to ensure a fair comparison. On SQL Server, we manually translate the optimized UDFs produced by PRISM from PL/pgSQL to Microsoft's T-SQL syntax. We then use SQL Server's "native compilation" feature to compile outlined UDFs to machine code [37]. We then inline the optimized UDF using SQL Server's implementation of FROID [42, 5], rewriting the UDF to use a single `RETURN` instruction as required to ensure that the DBMS performs inlining.

We performed our evaluation on a machine with a dual-socket 20-core Intel Xeon Gold 5218R CPU (20 cores per CPU, 2× HT), 192 GB DDR4 RAM, and a 960 GB NVMe SSD. We ran DuckDB on a single-socket with 32 threads to avoid a known bug with non-terminating queries [6] and use 80 threads for the maximum degree of parallelism (`maxDOP`) on SQL Server. We use the default index configuration for all workloads, and build additional column-store indexes [32] on every table on SQL Server. For each DBMS, we tune their configuration knobs to improve performance, pre-warm the buffer pool, and refresh statistics. We perform two warmup runs of each query and then five hot runs (with minimal observed variance), reporting the average execution time of the five runs.

5.1 Workloads (SQL ProcBench)

Microsoft released the SQL ProcBench in 2021 [19] as the first UDF-centric benchmark modeled after real-world UDFs on Azure SQL Server. ProcBench is based on the TPC-DS benchmark and contains 24 queries that invoke scalar UDFs. We use a scale factor of 10 (≈10 GB). We run 17 of the 24 queries, ignoring queries that use table-valued functions (TVFs) or UDFs invoked from stored procedures. We also skip queries Q8 and Q14 on SQL Server as these queries invoke UDFs with cursor loops. Aggify rewrites these cursor loops to custom aggregate functions that we could not compile on the Linux version of SQL Server 2022.

5.2 Single-Threaded Performance

We now evaluate DuckDB and SQL Server on single threaded execution with ProcBench. We run with a single thread to control for the effect of parallelism. We show in Section 5.3 that we observe similar results when running multi-threaded. ProcBench contains complex UDFs that inlining translates to subqueries with `LATERAL` joins. Therefore, ProcBench stresses PRISM's ability to simplify UDFs and produce easy-to-unnest, fast-to-execute queries.

DuckDB: The results in Figure 7a show DuckDB's performance improvement with PRISM. We use a log-scale axis for queries that are at least 10× faster (i.e., Q14 and Q17), and a linear scale for the remaining queries. Q14 is 18× faster with PRISM, as the UDF contains a loop that is faster to execute as a compiled outlined function compared to an inlined recursive CTE. Q17 executes 2270.8× faster with PRISM than the inlined UDF because the latter introduces a slow cross-product join into the query plan during unnesting. The cross-product generates 250 billion tuples which enter the probe side of the hash join, taking over 56 minutes to finish. By comparison, PRISM optimizes the UDF, eliminating all `LATERAL` joins and subqueries, avoiding the cross-product and executing orders of magnitude faster (terminating in ≈1.5s).

For the remaining queries, PRISM provides a speedup of 1.02–1.62× (on average 1.29×). After inspecting the query plans with `EXPLAIN ANALYZE`, we find the extra runtime is due to executing additional join operators introduced when DuckDB unnests subqueries and `LATERAL` joins. By comparison, PRISM eliminates these subqueries and joins, resulting in faster query plans.

SQL Server: We next measure PRISM's effect on SQL Server. We report the speedup of PRISM over the baseline in Figure 7b rather than reporting absolute numbers. We follow the same convention as above, reporting queries with over 10× improvement using a log-scale and the remaining queries with a linear scale. Our first observation is that most ProcBench queries (eight out of 15) are at least an order of magnitude (on average 559.1×) faster with PRISM relative

Table 1: ProcBench Subquery Unnesting – For each system (FROID, PRISM), applied to each DBMS (SQL Server, DuckDB), the table indicates whether the DBMS unnests all subqueries in a given Bench query (i.e., replaces them with join operators).

	Technique	ProcBench Queries Q1 Q5 Q6 Q7 Q9a Q9b Q12 Q13 Q15 Q17 Q18 Q20a1 Q20a2 Q20b1 Q20b2
SQL Server	Inlining	✗ ✗ ✗ ✗ ✓ ✓ ✗ ✗ ✗ ✗ ✗ ✓ ✓ ✗ ✗
	PRISM	✓ ✓ ✗ ✓ ✓ ✓ ✓ ✓ ✗ ✓ ✗ ✓ ✓ ✓ ✓
DuckDB	Inlining	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
	PRISM	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

to the baseline. The source of this improvement is that SQL Server cannot unnest the subqueries generated by inlining but unnests PRISM's simplified UDFs. Through unnesting, the DBMS replaces inefficient, iterative subqueries with set-oriented join operators that provide an algorithmic advantage and faster query performance. Our analysis of query plans for the ProcBench (Table 1) shows that SQL Server only unnests four out of 15 of FROID's inlined queries, compared to 12 with PRISM. Contrast this with DuckDB, which unnests all ProcBench queries [38, 40]. By simplifying PRISM's generated UDFs and hiding as much code as possible through outlining, SQL Server unnests the simplified queries, resulting in orders of magnitude faster query plans. For the remaining queries, PRISM is faster by avoiding unnecessary `LATERAL` joins (on average 1.12× faster). The exception is `Q18`, which is slower with PRISM due to duplicating `SELECT` statements.

5.3 Multi-Threaded Performance

To evaluate whether PRISM's benefits generalize to a multi-threaded setting, we perform the same measurements as in Section 5.2 but with the maximum number of threads. To avoid a known bug in DuckDB that prevents queries from terminating [6], we use a maximum of 32 threads in our DuckDB experiments.

DuckDB: Comparing single-threaded and multi-threaded performance (Figures 7a and 8a), PRISM provides similar improvements. We attribute DuckDB's stable scalability to its execution of physical operators in both query plans with HyPer-style morsel-based parallelism and scheduling [35].

SQL Server: Unlike DuckDB, PRISM's performance improvements are lower on SQL Server when executing queries multi-threaded. We attribute this gap to an implementation detail of SQL Server that forces single-threaded query plans when invoking non-inlined UDFs [42] (i.e., the outlined UDF pieces generated by PRISM). Hence, PRISM has a diminished benefit on SQL Server with multiple threads. SQL Server could address this problem by providing a `PARALLEL SAFE` annotation (similar to PostgreSQL), allowing outlined UDF pieces and the entire query plan to run in parallel.

6. RELATED WORK

We now discuss prior work related to our approach. We refer the reader to Section 2 for our overview of existing UDF optimization techniques (i.e., compilation, batching, and inlining).

Subquery Unnesting: [29] introduced the first technique to unnest subqueries. [44] extended their work by providing unnesting rules for more complex subqueries. SQL Server models subqueries using the `APPLY` operator, relying on rewrite rules to remove the `APPLY` operator from the query plan [16, 8]. However, this approach cannot unnest arbitrary queries [38]. In the context of UDF inlining, [15] demonstrated that SQL Server fails to unnest inlined UDFs when the generated subquery contains nested `APPLY/LATERAL` operators. PRISM builds on this observation, using UDF outlining to generate `LATERAL`-free subqueries wherever possible, enabling SQL Server to unnest many more inlined UDFs (see Table 1) and achieve significantly improved performance.

[38] pioneered the first algorithm to unnest arbitrary subqueries (which DuckDB implements [40]), allowing the DBMS to unnest arbitrary inlined UDFs. Our experiments found that for some UDFs (i.e., `Q17` from the ProcBench), unnesting the generated subquery results in orders of magnitude slower performance than naively evaluating the subquery row-by-row. Further research is necessary to unnest arbitrary subqueries without introducing regressions.

Optimizing Database-Backed Applications: A related research topic is lifting application logic into the DBMS. [3] use program synthesis to translate application code into SQL, reducing data movement between the application and the DBMS, and enabling the query optimizer to find better plans. [50] also explored program synthesis techniques to lift UDF code to SQL. Unfortunately, synthesis-based techniques lack termination guarantees, challenging their adoption in commercial systems [42].

In contrast, [10, 9] developed EqSQL, which uses static analysis techniques to translate application code into equivalent relational operations using a functional IR. Although EqSQL significantly improves performance, FROID chose a different translation strategy, which generates simpler code [42]. More recently, PyFROID [25, 36, 11] uses inlining-inspired techniques to translate queries written in pandas to SQL for faster query execution.

Cross-Language Optimization: Recent academic prototypes blur the lines between database systems and compilers. For instance, LingoDB [28, 27] uses MLIR [33] to represent relational and non-relational code as MLIR dialects, allowing cross-boundary optimizations. [17] adopt a similar approach, using GraalVM [49] to represent and optimize polyglot queries (i.e., queries invoking UDFs in multiple programming languages). Our techniques complement these designs, where the DBMS applies PRISM's optimizations on its unified IR.

7. CONCLUSION

In this paper, we demonstrated how *UDF outlining* improves performance relative to conventional UDF inlining by selectively inlining only the portions of the UDF that are helpful for query optimization. By combining UDF outlining with four other complementary UDF-centric optimizations, our implementation (PRISM) achieves substantial speedups for UDF-centric queries running on DuckDB and SQL Server due to three benefits: more effective unnesting (sometimes resulting in over a 1000× speedup), improved data skipping (resulting in roughly a 10× speedup), and avoiding unnecessary joins (typically resulting a 1.02–1.62× speedup).

8. REFERENCES

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[2] F E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[3] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices*, 48(6):3–14, 2013.

[4] R. Cytron et al. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT POPL symposium*, pages 25–35, 1989.

[5] E. Darling. When Does Scalar UDF Inlining Work In SQL Server? `https://erikdarlingdata.com/when-does-udf-inlining-kick-in/`, may 2022.

[6] DuckDB. Group by never completes #9718. `https://github.com/duckdb/duckdb/issues/9718`, november 2023.

[7] DuckDB. Overview of DuckDB Internals. `https://duckdb.org/docs/internals/overview.html`, april 2024.

[8] M Elhemali et al. Execution strategies for sql subqueries. In *Proceedings of the 2007 ACM SIGMOD Conference*, pages 993–1004, 2007.

[9] K Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S Sudarshan. Dbridge: Translating imperative code to sql. In *Proceedings of the 2017 ACM Conference*, pages 1663–1666, 2017.

[10] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1781–1796, San Francisco California USA, June 2016. ACM.

[11] Venkatesh Emani, Avrilia Floratou, and Carlo Curino. Pyfroid: Scaling data analysis on a commodity workstation. 2024.

[12] Steven Feuerstein. Speed up execution of your functions inside SQL statements with UDF pragma . `https://stevenfeuersteinonplsql.blogspot.com/2017/03/speed-up-execution-of-your-functions.html`, march 2017.

[13] Y Foufoulas, A Simitsis, and Y Ioannidis. YeSQL: rich user-defined functions without the overhead. *Proc. VLDB Endow.*, 15(12):3730–3733, August 2022.

[14] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. YeSQL: "you extend SQL" with rich and highly performant user-defined functions in relational databases. *Proc. VLDB Endow.*, 15(10), June 2022.

[15] Kai Franz, Samuel I Arch, Denis Hirn, Torsten Grust, Todd Mowry, and Pavlo. 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, Conference on Innovative Data Systems Research*, 2024.

[16] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. *ACM SIGMOD Record*, 30(2):571–581, 2001.

[17] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. Babelfish: efficient execution of polyglot queries. *Proc. VLDB Endow.*, 15(2):196–210, October 2021.

[18] S Gupta, S Purandare, and K Ramachandra. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 559–573. ACM, June 2020.

[19] S Gupta and K Ramachandra. Procedural extensions of SQL: understanding their usage in the wild. *Proc. VLDB Endow.*, 14(8):1378–1391, April 2021.

[20] Ravindra Guravannavar and S Sudarshan. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment*, 1(1):1107–1123, 2008.

[21] Matthew S Hecht and Jeffrey D Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, 1972.

[22] D Hirn and T Grust. One WITH RECURSIVE is Worth Many GOTOs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 723–735, June 2021.

[23] Denis Hirn. Decorrelation and parallelization of recursive and materialized CTEs #10357. `https://github.com/duckdb/duckdb/pull/10357`, feb 2023.

[24] Denis Hirn. Apfel. `https://apfel-db.cs.uni-tuebingen.de/`, may 2024.

[25] A Jindal et al. Magpie: Python at Speed and Scale using Cloud Backends. 2021.

[26] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, 1994.

[27] Michael Jungmair and Jana Giceva. Declarative sub-operators for universal data processing. *Proceedings of the VLDB Endowment*, 16(11):3461–3474, 2023.

[28] Michael Jungmair, André Kohn, and Jana Giceva. Designing an open framework for query optimization and compilation. *Proceedings of the VLDB Endowment*, 15(11):2389–2401, 2022.

[29] Won Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.

[30] Steffen Kläbe, Bobby DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. Accelerating python udfs in vectorized query execution. CIDR Conference, 2022.

[31] T Kotzmann et al. Design of the java hotspot™ client compiler for java 6. *ACM TACO*, 5(1):1–32, 2008.

[32] P-Å Larson et al. Sql server column store indexes. In *Proceedings of the 2011 ACM SIGMOD Conference*, pages 1177–1184, 2011.

[33] C Lattner et al. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM CGO*, pages 2–14, 2021.

[34] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[35] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the

many-core age. In *Proceedings of the 2014 ACM SIGMOD Conference*, pages 743–754, 2014.

[36] X Liu, V Emani, A Floratou, J Cahoon, P Seamark, and C Curino. Polysem: Efficient polyglot analytics on semantic data. 2023.

[37] Microsoft. Scalar User-Defined Functions for In-Memory OLTP. `https://learn.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/scalar-user-defined-functions-for-in-memory-oltp/`, june 2016.

[38] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[39] Oracle. Oracle9i Database New Features. `https://docs.oracle.com/cd/A91202_01/901_doc/server.901/a90120/ch2_feat.htm`, june 2001.

[40] Mark Raasveldt. Correlated Subqueries in SQL. `https://duckdb.org/2023/05/26/correlated-subqueries-in-sql.html`, may 2023.

[41] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 SIGMOD Conference*, pages 1981–1984, 2019.

[42] K Ramachandra, K Park, K. V Emani, A Halverson, C Galindo-Legaria, and C Cunningham. Froid: optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11(4):432–444, December 2017.

[43] B K Rosen, M N Wegman, and F K Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT POPL*, 1988.

[44] P Seshadri, H Pirahesh, and TY C Leung. Complex query decorrelation. In *Proceedings of the Twelfth ICDE*, pages 450–458, 1996.

[45] V Simhadri, K Ramachandra, A Chaitanya, R Guravannavar, and S. Sudarshan. Decorrelation of user defined function invocations in queries. In *2014 IEEE 30th ICDE*, pages 532–543, Chicago, IL, USA, March 2014. IEEE.

[46] SingleStore. SingleStoreDB Cloud Release Notes. `https://docs.singlestore.com/cloud/release-notes/singlestoredb-cloud-release-notes/`, november 2021.

[47] L Spiegelberg et al. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 SIGMOD Conference*, pages 1718–1731. ACM, June 2021.

[48] L F. Spiegelberg and T Kraska. Tuplex: robust, efficient analytics when Python rules. *Proc. VLDB Endow.*, 12(12):1958–1961, August 2019.

[49] T Würthinger et al. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

[50] G Zhang, Y Xu, X Shen, and I Dillig. UDF to SQL translation through compositional lazy inductive synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–26, October 2021.