The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining

Samuel Arch
Carnegie Mellon University
sarch@cs.cmu.edu

Yuchen Liu
Carnegie Mellon University
yuchenl6@andrew.cmu.edu

Todd C. Mowry
Carnegie Mellon University
tcm@cs.cmu.edu

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

Andrew Pavlo
Carnegie Mellon University
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 sub-optimal 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.

PVLDB Reference Format:
Samuel Arch, Yuchen Liu, Todd C. Mowry, Jignesh M. Patel, and Andrew Pavlo. The Key to Effective UDF Optimization:
Before Inlining, First Perform Outlining. PVLDB, 18(1): 1-13, 2024.
doi:10.14778/3696435.3696436

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SamArch27/PRISM.

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.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:10.14778/3696435.3696436

To address this impedance mismatch, researchers developed optimization techniques for UDFs, including *compilation* (i.e., generating specialized machine code for the UDF) [40, 43], *batching* (i.e., coalescing individual UDF invocations) [17, 22], and more recently, *UDF inlining* (i.e., translating a UDF into an equivalent SQL subquery) [26, 46, 49]. Inlining has shown the most potential of these techniques, providing up to 1000× performance improvements for workloads in a commercial DBMS [6].

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 [21] (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

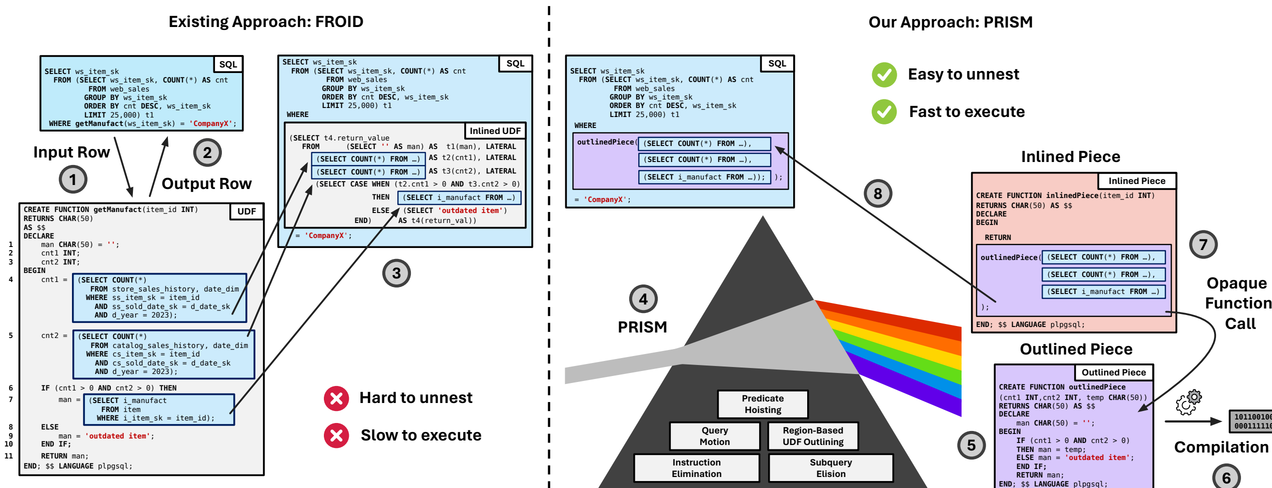


Figure 1: Overview of PRISM– The existing approach to UDF optimization (FROID) translates UDF execution from row-by-row oriented execution into a SQL subquery that it inlines into into the calling query. Our approach (PRISM) deconstructs a UDF into separate inlined and outlined UDF pieces. The framework 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.

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

We integrated PRISM into DuckDB [45] 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 [21] (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 [43], 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 [14] .

Microsoft SQL Server similarly added the ability to compile T-SQL UDFs to machine code in 2016 [40]. SingleStore added support to compile PL/pgSQL UDFs to LLVM IR in 2021 [50]. For dynamically-typed UDFs (i.e., Python UDFs), just-in-time (JIT) compilation is essential and is adopted by systems such as YeSQL [15, 16], BabelFish [19], Tuplex [51, 52], and Actian Vector [32]. 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 [22]. 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 [17]. Although this technique simplifies UDF-centric queries substantially, the overhead of materializing temporary tables is prohibitive and not scalable.

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 [49] (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 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` [46]. 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 [6].

Although FROID only supports inlining of loop-free UDFs, further techniques have since lifted this restriction. Aggify [20] 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 significant changes to the internals of the DBMS.

More recently, the **Apfel** framework provides a pure SQL translation for UDFs [26]. 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 that do not natively support UDFs.

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:

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 [17]. 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 [42, 45], 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 6, even for DuckDB, which supports arbitrary unnesting, these complex subqueries introduce additional join operators during the unnesting process, slowing down query performance.

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 their contents and identifying opportunities to avoid unnecessary work.

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 6, 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 joins in a query rather than making more of them.

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 a suite of 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

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

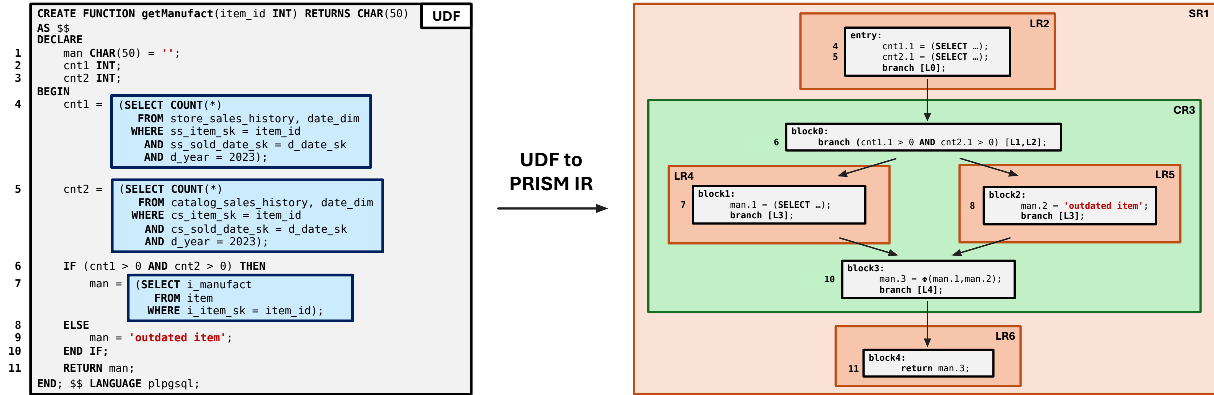


Figure 2: PRISM's Intermediate Representation (IR) – PRISM uses an SSA-based IR, ensuring variables are assigned exactly once in the entire program. The IR also encodes high-level structured control flow (conditionals, loops) as a hierarchy of program regions. **LR** denotes a leaf region, **CR** denotes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

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 (Section 4.5) 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 easier to unnest and faster.

3.2 Static Single Assignment (SSA) Form

PRISM represents UDFs as *control flow graphs* (CFGs) in *static single assignment* (SSA) form [5, 47] (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) [33, 36, 53].

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 [28], a hierarchy of single-entry single-exit (SESE) program *regions* [1, 23] (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) [1, 23, 28], 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. As we demonstrate in Section 6.7, 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 added.

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

Algorithm 1: Query Motion

```
Function QueryMotion(UDF f):
    changed ← True;
    while changed do
        changed ← False;
        worklist ← ∅;
        forall I ← f.getInstructions() do
            if I.containsSELECT() then
                worklist.insert(I);
        forall I ← worklist do
            vars ← I.getRHS().getUsedVariables();
            region ← I.getRegion();
            conds ← ∅;
            while vars ∩ region.getDefs() = ∅ do
                if region.isConditional() then
                    conds ← conds ∪ region.getCond();
                    vars ← vars ∪ region.getCond();
                region ← region.getParentRegion();
            if region ≠ I.getRegion() then
                changed ← True;
                I.hoistToRegion(region, conds);
    return changed;
```

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

PRISM achieves this by performing *query motion* using Algorithm 1. The idea is to repeatedly hoist SELECT statements as high as possible in the program until no further hoisting is possible. If the statement's variables do not depend on the current region, PRISM hoists the statement to the parent region. The process repeats until the current region defines at least one of the statement's variables. When hoisting through conditional regions, PRISM appends any predicates that guard these regions to the SELECT statement, ensuring that the query only executes when the predicate holds and no performance regressions occur.

Figure 3 illustrates the hoisting process for our motivating example. (1) Algorithm 1 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.

[2]Although PRISM performs its optimizations on its intermediate representation, we show the transformations on PL/pgSQL UDFs in our figures.

```
   CREATE FUNCTION getManufact(item_id INT) RETURNS CHAR(50)
   AS $$
   DECLARE
1      man CHAR(50) = '';
2     cnt1 INT;
3     cnt2 INT;
4    temp CHAR(50);  (2)
   BEGIN
5      cnt1 = (SELECT COUNT(*) FROM …);
6      cnt2 = (SELECT COUNT(*) FROM …);
7      temp = (SELECT (SELECT i_manufact FROM …)
                WHERE cnt1 > 0 AND cnt2 > 0);   (3)
8      IF (cnt1 > 0 AND cnt2 > 0) THEN
9          man = temp;  (SELECT i_manufact
                          FROM item
10     ELSE    (4)        WHERE i_item_sk = item_id);  (1)
11         man = 'outdated item';
12     END IF;
13     RETURN man;
   END; $$ LANGUAGE plpgsql;
```

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

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 we will show in Section 6.4). 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 [46], 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.

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 [1, 23], 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).

To maximize the amount of extracted code with region-based UDF outlining, PRISM employs a recursive algorithm (Algorithm 2)

Algorithm 2: Region-Based UDF Outlining

```
Function udfOutlining(UDF f, R region):
    if not region.containsSELECT() then
        outlineBlocks(f, region.getBlocks());
    else
        if not region.isSequential(); then
            forall subregion ← region.getSubregions() do
                udfOutlining(f, subregion);
        else
            Q queuedBlocks ← ∅;
            forall subregion ← region.getSubregions() do
                if not subregion.containsSELECT() then
                    queuedBlocks.insert(subregion.getBlocks());
                else
                    outlineBlocks(f, queuedBlocks);
                    queuedBlocks.clear();
                    udfOutlining(f, subregion);
            outlineBlocks(f, queuedBlocks);
```

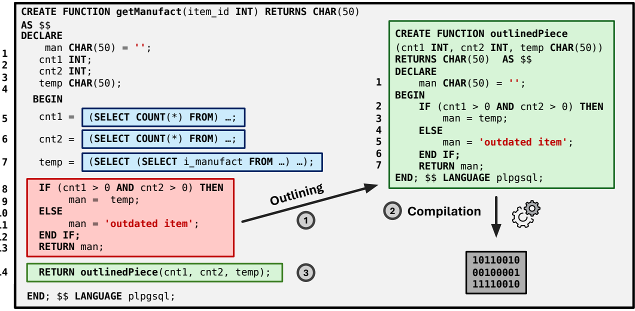


Figure 4: Region-Based UDF Outlining – Finding the UDF's largest procedural code regions and extracting them into functions for compilation.

that traverses the UDF's regions in a top-down manner. If a region does not contain a SELECT, the algorithm outlines all its blocks (including those of its subregions) into a new function. Otherwise, the algorithm considers the subregions, attempting to outline as much code as possible. If the region is not sequential, Algorithm 2 is called recursively on the subregion. PRISM treats sequential regions differently by outlining the longest sequence of SELECT-free regions together (since SELECT statements are guaranteed to start a new region). The algorithm achieves this by maintaining a queue of basic blocks and appends each region's blocks to the queue until it encounters a region containing a SELECT statement. At this point, the algorithm flushes the queue of basic blocks, outlining them into a new function, and the algorithm continues.

Figure 4 illustrates the outlining process for our example UDF. First, ① Algorithm 2 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 [20]. 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). 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 [9] and SQL Server [46]) to identify the duplicated expression during query optimization and evaluate it 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 [42] (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 (discussed in Section 6.5) require PRISM to decide whether to duplicate SELECT statements.

Algorithm 3 contains PRISM's instruction elimination method. If an instruction defined by a SELECT has multiple uses and the DBMS unnests arbitrary queries [42], 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) to ensure termination.

Figure 5 illustrates the instruction elimination process for the motivating example. First, PRISM replaces the uses of ① *cnt1* on

Algorithm 3: Instruction Elimination

```
Function InstructionElimination(UDF f):
    changed ← True;
    while changed do
        changed ← False;
        worklist ← f.getInstructions();
        forall I ← worklist do
            U ← I.getUses()
            if I ∈ U then
                continue;
            if U.size() > 1 and I.getRHS().isSELECT() then
                if DBMS.canUnnestArbitraryQueries() then
                    continue;
            forall U ← I.getUses() do
                changed ← True;
                U.replaceUsesWith(I.getLHS(), I.getRHS());
                worklist.insert(U);
    return changed;
```

lines 6–7 with its definition and then ② does the same for *cnt2*. Next, ③ Algorithm 3 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.

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, sidestepping the inlining process entirely 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.

4.5 Predicate Hoisting

The last challenging scenario for UDF optimization is when queries invoke a UDF in their `WHERE` clause. These queries are problematic because the DBMS's optimizer cannot push UDF predicates down into the leaves of a query plan for data-skipping. Without such data-skipping optimizations, the DBMS falls back to scanning entire tables and evaluating the UDF predicate for each row.

There are two main reasons why these UDFs are difficult to automatically optimize. Foremost is that these predicates often use `IF/ELSE` blocks to filter tuples, which are not always boolean functions. For example, the query in our example in Figure 1 checks whether the UDF's return value is equal to the string `'CompanyX'`. The second reason, is that existing optimizations obfuscate such

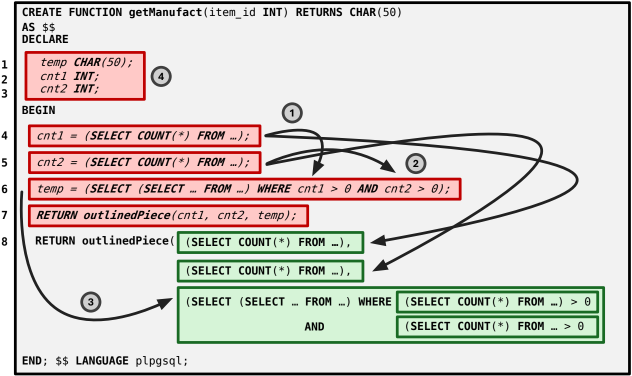


Figure 5: Instruction Elimination – Replacing each use of a variable with its definition, eliminating the instructions in a UDF, collapsing it down to a single `RETURN` instruction.

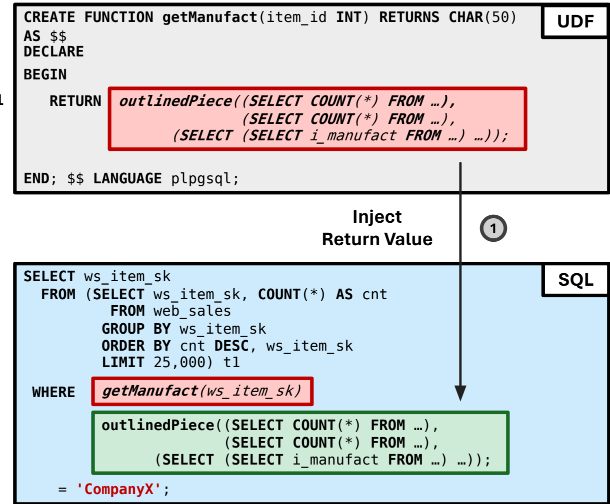


Figure 6: Subquery Elision – Replacing the UDF call with its return value, bypassing UDF inlining and the corresponding subquery.

predicates: the `IF/ELSE` logic is either hidden from the optimizer with UDF outlining or muddled with `LATERAL` joins after inlining.

PRISM overcomes these challenges with *predicate hoisting* by analyzing a UDF and then expressing it as a boolean predicate that injects into a query's `WHERE` clause. As shown in Algorithm 4, PRISM first identifies all program paths to each `RETURN` instruction. It then converts each path through the program into a conjunctive predicate and disjuncts the predicates from each path together. PRISM resolves each predicate's variables by following use-def chains before conjuncting them together. PRISM expresses each path as a boolean predicate by generating a predicate *new* that compares the return value with an unseen parameter t using an operator `<op>` that the DBMS could exploit for data skipping (i.e., $=, <, \leq, >, \geq$). Then, for every condition that it encounters along the path to the `RETURN`, PRISM conjuncts the condition with *new*. For the five possible choices of `<op>`, PRISM generates a boolean UDF. When a query invokes the UDF from the `WHERE` clause, PRISM substitutes the corresponding predicate in place of the original UDF call. PRISM's predicate hoisting implementation ignores UDFs with loops (since they are not straightforward to express as filters). Although PRISM

Algorithm 4: Predicate Hoisting

```
Function PredicateHoisting(UDF f):
    if f.containsLoop() then
        return
    pred ← True;
    forall I ← f.getInstructions() do
        if I.isReturnStatement() then
            forall P ← getPathsToReturn(I) do
                new ← (I.getRHS() <op> t)
                new ← new ∧ [⋀ getCondsOnPath(P)]
                pred ← pred ∨ resolveVarsToArguments(new)
    return pred;
```

could exclusively outline loop regions before predicate hoisting, this is outside of the scope of PRISM and we defer it as future work.

Figure 7 illustrates our approach for FROID's TPC-H Q1 UDF example [46]. ① PRISM rewrites the UDF to return a `BOOLEAN` instead of an `INT` value. Next, PRISM adds an additional input argument to the UDF: the t variable is the same type as the function's original return value (`INT`). Then, ③ PRISM translates the UDF into an equivalent predicate (disjuncting the two paths through the UDF that reach the `RETURN` instructions on lines 4 and 6). Next, ④ the system rewrites the query's `WHERE` clause to pass the compared value `1` as the new parameter t to the UDF. Lastly, ⑤ PRISM substitutes the boolean predicate into the `WHERE` clause and then ⑥ the DBMS's optimizer performs logical rewrites to simplify the predicate.

For the query shown in Figure 7, predicate hoisting improves performance by 21.7× on DuckDB and 50.8× on SQL Server.

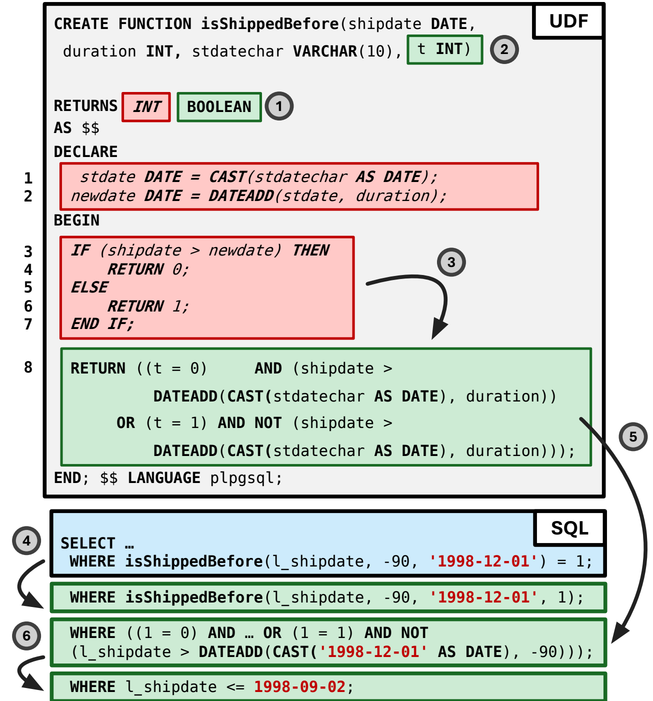


Figure 7: Predicate Hoisting – Analyzing loop-free UDF predicates to rewrite them as boolean formulae substituted into the `WHERE` clause, making data-skipping optimizations transparent to the query optimizer.

5 IMPLEMENTATION

We now discuss details about our implementation.

Nested Lateral Joins in DuckDB: We encountered binder errors with DuckDB when executing PRISM's inlined UDFs, as the DBMS lacked support for nested queries containing `LATERAL` joins. Unfortunately, this feature is needed to implement UDF inlining. To overcome this challenge, we implemented nested `LATERAL` joins in DuckDB [3]. We modified DuckDB's binder to correctly bind, plan, and unnest `LATERAL` joins in subqueries. Our patch [3] to add UDF inlining to DuckDB was released in DuckDB v0.9 [41].

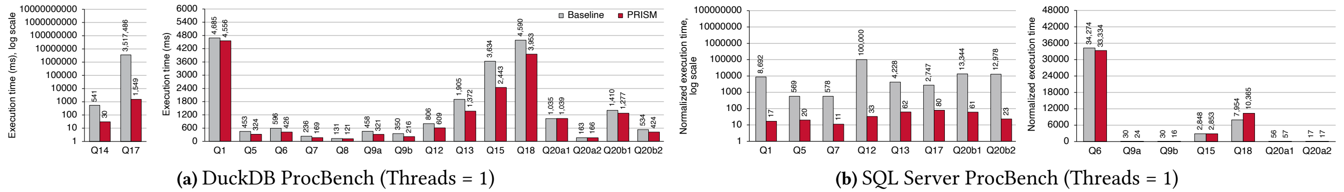
Integration into DuckDB: We implemented PRISM as a 7000-line C++ statically linked extension for DuckDB. It includes a PL/pgSQL front-end, intermediate representation, and code generator for compiled code fragments. When a user issues a `CREATE FUNCTION` command from the DuckDB shell, our extension intercepts it and uses `libpgquery` [7] to parse the UDF into an abstract syntax tree (AST). Next, PRISM translates the AST into its IR using DuckDB's binder to resolve expressions and table aliases from the catalog. During this process, PRISM create a temporary table in DuckDB's catalog containing the UDF's variables as columns, enabling expressions that refer to program variables to bind correctly. PRISM compiles outlined UDF pieces to machine code by transpiling them to C++ functions. Transpilation occurs by walking the IR, resolving each bound DuckDB expression to a corresponding C++ function using the catalog, and emitting the code as part of a new DuckDB extension. Our system then invokes `clang` to compile and dynamically link the new extension into the DBMS. Our implementation inlines predicates or expressions as DuckDB macros. PRISM then inlines UDFs containing multiple instructions using Apfel [25].

Fixing a Bug in Aggify: Cursor loops in UDFs are common and prevent the DBMS from inlining them. Microsoft introduced **Aggify** in 2020 to rewrite cursor loops into equivalent custom aggregate functions that are eligible for inlining [20]. To support UDFs containing cursor loops (i.e., UDFs 8 and 14 from ProcBench [21]), we implemented Aggify in PRISM. But we encountered a bug where UDFs containing cursor loops produced incorrect results when the cursor loop executes for zero iterations. The original Aggify algorithm returns `NULL` in this case, which is incorrect when the result set is empty. Thus, we modified Aggify's algorithm to introduce an explicit check (via a `CASE` statement) in the rewritten UDF.

Permissions & Security: Although our implementation of PRISM in DuckDB does not check permissions or consider security, a commercial DBMS could safely adopt our techniques. PRISM combines two features, UDF inlining, and UDF compilation, that already exist in commercial DBMSs. By relying on these existing DBMS features, PRISM could be safely integrated and ensure correctness with respect to permissions and security.

6 EXPERIMENTAL ANALYSIS

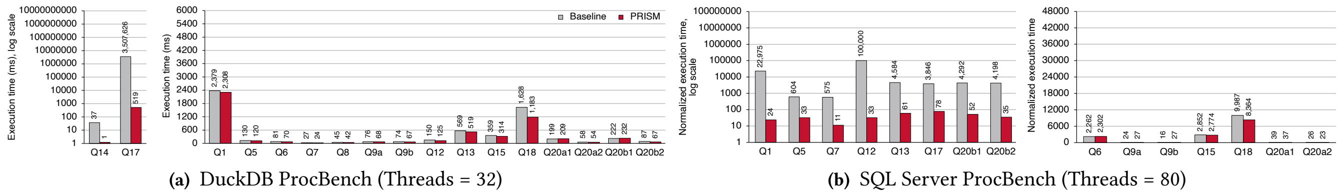
We now present an evaluation of PRISM in DuckDB [45] (commit `53dc13d` with a patch applied to support parallel CTEs [24]) and Microsoft SQL Server 2022. By integrating PRISM into DuckDB, the DBMS natively executes UDFs after registering them with the `CREATE FUNCTION` command. To compare against inlining, we used Apfel [25] 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



(a) DuckDB ProcBench (Threads = 1)

(b) SQL Server ProcBench (Threads = 1)

Figure 8: 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 9: 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 [8], 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.

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 [40]. We then inline the optimized UDF using SQL Server's implementation of FROID [6, 46], 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 [8] 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 [34] 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.

6.1 Workloads

We first describe the workloads that we use in our evaluation:

SQL ProcBench: Microsoft released the SQL ProcBench in 2021 [21] 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.

TPC-H FROID: We also evaluate FROID's variant of TPC-H that manually rewrites 12 queries from the benchmark to use scalar UDFs. These UDFs have interesting properties that are relevant to our analysis. First, the UDFs are simpler than in ProcBench. For example, TPC-H UDFs are purely procedural or relational, whereas ProcBench mixes procedural code and `SELECT` statements. They also factor out expressions into reusable functions rather than complex business logic. Lastly, the queries invoke UDFs from the `WHERE` clause to filter tuples, thereby blocking data skipping optimizations unless the DBMS uses predicate hoisting.

Table 1: Overall Speedup of PRISM (Single-Threaded) – Average and maximum speedup of PRISM for each DBMS, benchmark combination.

	Avg Speedup (Without Outliers)	Max Speedup (With Outliers)
DuckDB (ProcBench)	1.29×	2270.22×
SQL Server (ProcBench)	298.73×	2997.92×
DuckDB (TPC-H)	17.69×	43.68×
SQL Server (TPC-H)	135.83×	1053.32×

6.2 Overall Speedup

Our first experiment seeks to determine the overall benefit that PRISM achieves for UDF-centric queries. We measure the average and maximum speedup of PRISM on the ProcBench and TPC-H benchmarks on DuckDB and SQL Server running on a single CPU thread. Although PRISM improves overall performance significantly (often by orders of magnitude), there are "outlier" queries that provide a much larger benefit than others. We report average speedups without these outliers to provide a more accurate assessment.

The results in Table 1 show the average and maximum speedups with PRISM described above. DuckDB delivers a 1.29× speedup on ProcBench by eliminating `LATERAL` joins and subqueries (and their corresponding hash joins after unnesting). The maximum speedup of 2270.22× is due to DuckDB introducing a large cross-product into the query plan during unnesting. SQL Server improves by an average of 298.73× over inlining since PRISM generates UDFs that the DBMS unnests and evaluates efficiently with joins. The maximum

Table 2: 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	Q1	Q5	Q6	Q7	Q9a	Q9b	Q12	Q13	Q15	Q17	Q18	Q20a1	Q20a2	Q20b1	Q20b2
SQL Server	**Inlining**	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
	PRISM	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
DuckDB	**Inlining**	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	PRISM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

speedup is 2997.92× by replacing an expensive subquery with a hash join. For TPC-H, DuckDB delivers an average 17.69× speedup through better data skipping and avoiding `LATERAL` joins. The maximum speedup is 43.68× and 1053.32× by exposing a low-selectivity predicate to the optimizer in TPC-H Q12 and Q6, respectively. SQL Server delivers a 135.83× speedup for TPC-H by exploiting data skipping and "batch mode" predicate execution [34].

6.3 Single-Threaded Performance (ProcBench)

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 6.6 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 ProcBench: The results in Figure 8a 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. We study the performance improvement for Q14 in Section 6.4. 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 $\approx 1.5s$).

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 ProcBench: We next measure PRISM's effect on SQL Server. We report the speedup of PRISM over the baseline in Figure 8b 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 to the baseline. The source of this improvement

Table 3: Hardware Counters for ProcBench Q14 (Single-Threaded) – We use PerfEvent [37] to measure hardware-level counters, reporting their values for Q14 after inlining and compilation after outlining with PRISM.

	Inlined	Compiled	Ratio
Execution Time (ms)	3,707	234	15.82×
Instructions Retired	9,727,134,114	2,103,326,192	4.62×
Branches	1,711,551,933	513,643,284	3.33×
Branch Mispredicts	5,426,332	789,462	6.87×
LLC Misses	215,522,377	4,134,696	52.12×

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 2) 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 [42, 44]. 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. We discuss the trade-offs of such duplication in Section 6.5.

6.4 Case Study: ProcBench Q14

To understand why Q14 is an order of magnitude faster with PRISM compared to the baseline, we perform a microarchitectural analysis of the query performance, instrumenting DuckDB to collect hardware counters with PerfEvent [37]. We run Q14 one million times with random input data first with inlining and then again as a compiled function using PRISM. The performance gap arises because Q14 contains a loop that inlining converts into a recursive CTE and `LATERAL` joins, but PRISM outlines and compiles this loop.

Table 3 reports the recorded hardware counters from our experiment. First, the overall execution time is reduced by 15.82× with compilation due to a decreased instruction count (4.62× fewer retired instructions), and fewer branch mispredictions. However, the performance gap is most apparent when observing the number of last-level cache (LLC) misses, stalling the CPU 52.12× as often to fetch data into the CPU's caches. The number of cache misses increases due to the use of SQL features to simulate procedural constructs. The recursive CTE populates and clears working tables to emulate looping functionality, and the `LATERAL` joins are unnested and execute as hash joins. The DBMS incurs more LLC misses from building and accessing these intermediate structures. By comparison, PRISM outlines the UDF into a separate function and invokes an optimizing compiler (with its own loop and memory optimizations), translating the function to efficient machine code.

6.5 Query Duplication

For three of ProcBench's UDF-centric queries in our experiments (Q18, Q20b1, Q20b2), PRISM decides during instruction elimination (Section 4.3) whether to duplicate queries (i.e., `SELECT` statements). By performing query duplication, PRISM reduces the number of `LATERAL` joins but the DBMS will evaluate the `SELECT` statement

Table 4: **SQL Server ProcBench Query Duplication Execution** – Single-threaded execution times for ProcBench queries with query duplication disabled ("Dup. Off") and enables ("Dup. On"). We normalize execution times with the maximum value reported as 100,000.

	Execution Time (Dup. Off)	Execution Time (Dup. On)	Ratio (Off / On)
Q18	43,231	54,128	0.79×
Q20b1	74,917	100,000	0.75×
Q20b2	72,502	93,307	0.78×

multiple times in the query plan. With such duplication turned off, the DBMS evaluates the query once but the inlined UDF will contain LATERAL joins that may prevent unnesting for DBMSs that cannot unnest arbitrary queries (i.e., SQL Server).

To understand the effect of query duplication on performance, we measure the execution times for these three queries with and without the optimization enabled. Since we originally designed PRISM's query duplication technique for SQL Server, we use this DBMS for this experiment running on a single CPU thread. We report the queries' normalized execution time and relative performance ratio. We also inspect the optimized query plans to determine whether the DBMS unnests the generated subqueries.

The results in Table 4 show that the queries run 21–25% slower with duplication enabled. The reason for this slowdown is that eliminating the LATERAL joins with query duplication does not affect whether SQL Server unnests these three queries (shown in Table 5). Although query duplication results in slower execution time for this DBMS, we leave the optimization on by default in PRISM. The performance benefit of unnesting (switching to an efficient set-oriented query plan rather than a slow-iterative plan) almost always results in multiple orders of magnitude faster execution times, which far outweigh the much smaller overhead introduced with query duplication. After turning off query duplication, we found that PRISM always matches or outperforms inlining. Therefore, to avoid performance regressions, we recommend that commercial DBMSs should run PRISM with query duplication turned off.

6.6 Multi-Threaded Performance (ProcBench)

To evaluate whether PRISM's benefits generalize to a multi-threaded setting, we perform the same measurements as in Section 6.3 but with the maximum number of threads. To avoid a known bug in DuckDB that prevents queries from terminating [8], we use a maximum of 32 threads in our DuckDB experiments.

DuckDB ProcBench: Comparing single-threaded and multi-threaded performance (Figures 8a and 9a), 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 [38].

SQL Server ProcBench: 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 [46] (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.

Table 5: **SQL Server ProcBench Query Duplication** – Whether the DBMS unnests all subqueries in Table 4's queries (replaces them with join operators) with and without PRISM's duplication optimization.

		ProcBench Queries		
	Approach	Q18	Q20b1	Q20b2
SQL Server	**PRISM (Dup. Off)**	✗	✓	✓
	PRISM (Dup. On)	✗	✓	✓

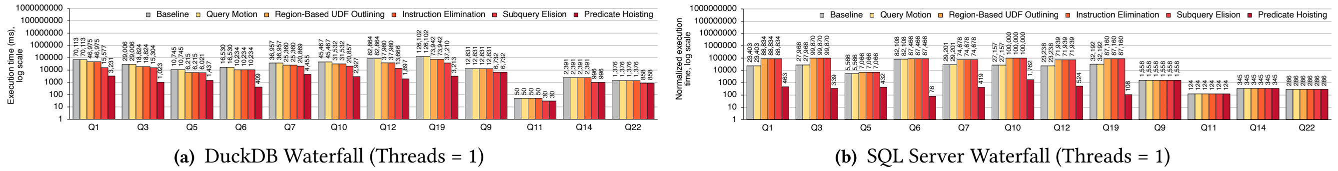
6.7 Progressive Optimization (TPC-H)

Lastly, we consider the impact of incrementally applying each of PRISM's optimizations using the TPC-H workload. As before (see Section 6.2), we run all queries single-threaded to ensure a fair comparison between inlining and PRISM.

DuckDB TPC-H: Figure 10a shows PRISM's performance gains with each added optimization. The bars are listed from left to right, corresponding to each optimization. The leftmost bar represents the baseline (no optimizations), with each bar incrementally applying an additional optimization until the rightmost bar with all optimizations enabled. The leftmost eight queries invoke UDFs from SELECT and WHERE clauses, whereas the rightmost queries (Q9, Q11, Q14, Q22) only invoke UDFs from the SELECT clause.

Our first observation is that query execution times decrease as PRISM applies each optimization, indicating that each technique benefits overall performance. Additionally, the performance gap between no optimizations (i.e., baseline) and full optimizations is stark, ranging from 1.6–43.68× (with an average speedup of 17.69×). Considering the optimizations in order, we observe that **query motion** does not affect query performance since the TPC-H UDFs do not mix relational and procedural code. Region-based **UDF outlining** eliminates LATERAL joins by extracting as much code as possible into separate functions, improving performance by up to 2.18× (on average by 1.43×). **Instruction elimination** does not affect performance because the entire UDF code is outlined since it is purely procedural. Next, **subquery elision** yields an additional speedup of 1.78× by substituting the return value of the UDF into the calling query without generating a subquery, thereby saving a join operator in the unnested query plan. Lastly, **predicate hoisting** has a significant effect on the leftmost eight queries (as they invoke UDFs from the WHERE clause), providing an additional 9.94× speedup. PRISM injects equivalent predicates into the query's WHERE clause, enabling DuckDB to apply predicate pushdown and skip over irrelevant blocks with zone maps. In summary, PRISM's optimizations are complementary and progressively improve query performance by removing join operators from the query plan and making data skipping optimizations transparent to the DBMS.

SQL Server TPC-H: Figure 10b shows the impact of each optimization on SQL Server. As with DuckDB, **query motion** and **instruction elimination** have no effect, and **subquery elision** also has no effect (since FROID already supports this optimization). However, **UDF outlining** results in an average performance degradation of 2.71× after outlining UDF predicates in the eight leftmost queries (which predicate hoisting will address). We attribute this slowdown to SQL Server's implementation of column-store indexes [34] that enable the DBMS to push down expressions into scans. After pushdown, expressions are executed in "batch mode" (i.e., on a compressed input vector). However, SQL Server blocks



(a) DuckDB Waterfall (Threads = 1)

(b) SQL Server Waterfall (Threads = 1)

Figure 10: TPC-H Waterfall (Single-Threaded) – Single-threaded execution times for DuckDB and SQL Server for the TPC-H queries from the FROID paper [46]. Each bar represents the execution time for PRISM with each optimization progressively enabled. The leftmost bar represents the "Baseline" (no optimizations), with each bar applying an additional optimization until the last one with all optimizations enabled. The leftmost eight queries invoke UDFs from the `SELECT` and `WHERE` clauses, whereas the rightmost queries (Q9, Q11, Q14, and Q22) only invoke UDFs from the `SELECT` clause.

pushdown of compiled UDFs, forcing decompression, and row-oriented execution for the UDF.

At this point, the DBMS is not exploiting data-skipping optimizations, as the predicate is outlined as an opaque function call. But after PRISM applies **predicate hoisting**, these filters are now injected into the `WHERE` clause of the query, allowing SQL Server to exploit both batch-mode execution and data-skipping with zone maps, resulting in an average improvement of 350.25× for the eight leftmost queries using UDF predicates. PRISM has no effect on queries Q9, Q11, Q14, and Q22 because their UDFs are simple expressions invoked exclusively in the `SELECT` clause that FROID can inline. Overall, with PRISM's optimizations, SQL Server fully exploits the benefits of columnar storage, delivering multiple orders of magnitude faster query plans (on average 203.25× over the baseline), and for Q6 (which relies on a highly selective UDF predicate), achieves over 1000× better performance with PRISM over inlining.

We ran this same experiment on ProcBench but did not see the same incremental benefit. That is, each optimization has no effect until PRISM applies subquery elision, which then unlocks all the improvements. Only after subquery elision do the queries become simple to unnest and faster to execute (shown in Figure 8).

7 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: Kim [31] introduced the first technique to unnest subqueries. Seshadri et al. [48] 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 [10, 18]. However, this approach cannot unnest arbitrary queries [42]. In the context of UDF inlining, Franz et al. [17] 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 to unnest many more inlined UDFs (see Table 2) and achieve significantly improved performance.

Neumann and Kemper [42] pioneered the first algorithm to unnest arbitrary subqueries (which DuckDB implements [44]), 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 performance regressions.

Optimizing Database-Backed Applications: A related research topic is lifting application logic into the DBMS. Cheung et al. [4] 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. Zhang et al. [54] also explored program synthesis techniques to lift UDF code to SQL. Unfortunately, synthesis-based techniques lack termination guarantees, challenging their adoption in commercial systems [46].

In contrast, Emani et al. [11, 12] 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 [46]. More recently, PyFROID [13, 27, 39] 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 [29, 30] uses MLIR [35] to represent relational and non-relational code as MLIR dialects, allowing cross-boundary optimizations. Grulich et al. [19] adopt a similar approach, using GraalVM [53] 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.

8 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).

ACKNOWLEDGMENTS

This work was supported (in part) by the National Science Foundation (IIS-2404373), Google DAPA Research Grants, and the CMU-DB Industry Affiliates Program.

REFERENCES

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

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

[3] Samuel Arch Arham Chopra, Mayank Baranwal. 2023. Add support for nested laterals #7528. https://github.com/duckdb/duckdb/pull/7528.

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

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

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

[7] Inc. (pganalyze) Duboce Labs. 2024. C library for accessing the PostgreSQL parser outside of the server environment. https://github.com/pganalyze/libpg_query.

[8] DuckDB. 2023. Group by never completes #9718. https://github.com/duckdb/duckdb/issues/9718.

[9] DuckDB. 2024. Overview of DuckDB Internals. https://duckdb.org/docs/internals/overview.html.

[10] M Elhemali et al. 2007. Execution strategies for SQL subqueries. In *Proceedings of the 2007 ACM SIGMOD Conference.* 993–1004.

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

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

[13] Venkatesh Emani, Avrilia Floratou, and Carlo Curino. 2024. PyFroid: Scaling Data Analysis on a Commodity Workstation. (2024).

[14] Steven Feuerstein. 2017. 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.

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

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

[17] Kai Franz, Samuel I Arch, Denis Hirn, Torsten Grust, Todd Mowry, and Pavlo. 2024. Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In *CIDR 2024, Conference on Innovative Data Systems Research.*

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

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

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

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

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

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

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

[25] Denis Hirn. 2024. Apfel. https://apfel-db.cs.uni-tuebingen.de/.

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

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

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

[29] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3461–3474.

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

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

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

[33] T Kotzmann et al. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM TACO* 5, 1 (2008), 1–32.

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

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

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

[37] Viktor Leis. 2023. PerfEvent. https://github.com/viktorleis/perfevent.

[38] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD Conference.* 743–754.

[39] X Liu, V Emani, A Floratou, J Cahoon, P Seamark, and C Curino. 2023. PolySem: Efficient Polyglot Analytics on Semantic Data. (2023).

[40] Microsoft. 2016. 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/.

[41] Hannes Mühleisen. 2023. 0.9.0 Preview Release "Undulata". https://github.com/duckdb/duckdb/releases/tag/v0.9.0.

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

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

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

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

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

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

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

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

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

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

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

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

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