An Evaluation of Compilation-Based PL/PGSQL Execution

Tanuj Nayak

CMU-CS-21-101

February 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andy Pavlo (Chair)
Todd C. Mowry

Submitted in partial fulfillment of the requirements for the Fifth Year Master's Program.

Keywords: User Defined Functions, Compilation, Inlining

Abstract

User Defined Functions (UDFs) are an important analytical feature in modern Database Management Systems (DBMSs) due to their server-side execution properties. These properties allow complex analytical queries to execute without serializing intermediate data over a network. However, query engines often incur significant overheads when executing UDFs due to them being non-declarative in contrast to SQL queries. This contrast causes a lot of context switching between UDF and SQL execution. As a given UDF invokes more SQL queries, these overheads become more noticeable. In this thesis, we investigate the extent to which compilation allow us to overcome such overheads. Compilation for executing SQL queries has become popular in database research in the past decade, especially in the context of main memory DBMSs. It has been shown to deliver significant improvements to query execution performance. We compare the technique of compiling UDFs with query inlining, another recent UDF execution technique. To make this comparison, we implemented a UDF compilation framework in NoisePage, a main-memory compilation-based DBMS. In this framework we compile UDFs into a domain-specific language (DSL) function and evaluated it against query inlining. We find that this framework has greater support across UDF language features than inlining frameworks and allows for more efficient functions. We also observe that our framework compiles functions into DSL primitives that are far more fine-grained and lightweight than most SQL operators. As a result, the SQL operators produced by the inlining approach incur a much larger performance overhead. On iteration-heavy benchmarks, the database system achieved performance gains from 2x to 120x with compilation relative to inlining.

Acknowledgments

I would like to thank my advisor Andy Pavlo for his continued support and guidance during my last three years at CMU. He nurtured my curiosity in systems research and added an important dimension to my CMU experience. I would also like to thank my colleagues at the CMUDB research group who helped me with this project, including Prashanth Menon, Amadou Ngom, Soumil Mukherjee, Deepayan Patra, and Wan Shen Lim. Last but not least, I am always thankful for my family and friends who have been invaluable to me throughout my life.

Contents

8 Appendix

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction and Background

Database management systems (DBMSs) since their inception have been a cornerstone of data storage and analytics for many applications. Of late, the analytical component's value within SQL DBMSs has been undermined by higher-level compute frameworks such as Spark and MapReduce [19, 44]. These frameworks pull data from a DBMS lower on the stack and then perform computations on the serialized data after pulling them into this higher-level framework. This process incurs high costs when serializing data over a network or IPC buffer. These costs are avoidable, especially when performing analytical queries where the input could be gigabytes of data and the output is at most half a kilobyte of data. Performing all required computations at the data storage site (i.e., the DBMS server itself) avoids these costs. User-Defined Functions allow users to perform complex queries with this idea. This thesis discusses a new UDF execution framework that operates via compilation in a main-memory DBMS and compares this technique with query-inlining, another UDF execution framework.

1.1 User-Defined Functions

Early DBMSs in the 1970s supported third-party analytical subroutines by providing interfaces to the DBMS data from languages such as C, Pascal, Ada, or FORTRAN [2, 29]. Programs written in these host languages could pass query strings to the DBMS and receive serialized results to continue with the rest of the host language logic. DBMSs would also support triggers that were batches of SQL statements that executed upon database events, like an insertion into a table [4]. In the late 1980s Oracle developed a language called PL/SQL that would execute within the DBMS runtime itself [4]. This was one of the first procedural User-Defined Functions (UDF) languages that executed within the DBMS itself. Many other popular UDF languages came about such as Transact-SQL (T-SQL) from Sybase/SQL Server and PL/PGSQL from PostgreSQL [6, 10, 8].

User-Defined Functions (UDFs) are imperative logic blocks that a client provides and stores inside the DBMS server to invoke later as part of a query. These functions can take in parameters that could be rows or column values from tables provided by the underlying DBMS. UDFs are supported in most modern relational DBMSs [13, 17, 12]. They give a few interesting features

```
CREATE FUNCTION totalRevenue (customer INTEGER) RETURNS DECIMAL AS
$$
DECLARE
    sum DECIMAL := 0;
BEGIN
    sum = SELECT sum(O_TOTALPRICE) FROM ORDERS WHERE O_CUSTKEY =
    ↪  customer;
    RETURN sum;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 1.1: Example of UDF that returns the sum of total prices for all tuples whose `O_CUSTKEY` is equal to the input parameter from the TPC-H `orders` table

```
SELECT C_CUSTKEY, totalRevenue(C_CUSTKEY) FROM CUSTOMER;
```

Listing 1.2: Invocation of Listing 1.1

that make research into them worthwhile. They give the modularity benefits traditional functions provide. Such modularity can be useful in workloads where a client may wish to invoke a particular analytical subroutine repeatedly. A client can also write these UDFs in languages such as T-SQL or PL/PGSQL, whose imperative properties provide an intuitive abstraction for writing analytical subroutines [42]. More importantly, the most attractive factor is that such subroutines are computed on the DBMS server where the data is stored, negating network serialization costs. Instead of serializing entire data tables or columns to a client, we have to serialize the aggregate results of a stored User Defined Function to the client, pushing the computational bottleneck from the network back to the server computation. Consider the example in Listing 1.7. Suppose the `orders` table contains 1 billion tuples. If this function was executing on the client-side, a DBMS would have to serialize all tuples of the query `SELECT O_TOTALPRICE FROM orders` to the client over a network. This is all to produce `sum` that is at most a 128-bit value. Instead, if the DBMS server executes this function on the server-side, the only network serialization needed is for `sum`. This way, UDFs avoid such large intermediary network serializations.

```sql
SELECT C_CUSTKEY, totalRevenue(C_CUSTKEY) FROM CUSTOMER;
```

Listing 1.3: An invocation of totalRevenue from Listing 1.1. Many optimizers do not read into the definition of `totalRevenue` and produce a suboptimal plan.

```sql
    SELECT c.C_CUSTKEY, SUM(O_TOTALPRICE) FROM CUSTOMER c JOIN ORDERS o
  ↪   ON c.C_CUSTKEY = o.O_CUSTKEY GROUP BY c.C_CUSTKEY HAVING
  ↪   SUM(O_TOTALPRICE) > 5000;
```

```sql
    SELECT C_CUSTKEY, totalRevenue(C_CUSTKEY) FROM CUSTOMER WHERE
  ↪   totalRevenue(C_CUSTKEY) > 5000;
```

```sql
    SELECT C_CUSTKEY, revenues.* FROM CUSTOMER c, LATERAL (SELECT
  ↪   SUM(O_TOTALPRICE) as total FROM ORDERS WHERE c.C_CUSTKEY =
  ↪   O_ORDERKEY) p, LATERAL (SELECT p.total WHERE p.total > 5000)
  ↪   revenues;
```

An important observation to note from such UDFs is that their imperative nature differs from that of SQL queries. SQL consists of declarative statements with no specified control flow, hence is known as a declarative language. This is different from the line-by-line nature of Listing 1.1. Despite this semantic disconnect between UDFs and SQL queries, major DBMSs still need to support the two features. They have often done so by having two separate frameworks for the two language paradigms. Hence, when we need the two languages to work in conjunction with each other, such as in a UDF that frequently calls SQL statements inside it, DBMSs run into context switching overheads [21, 42]. Another source of inefficiency in UDF execution lies in the inability of traditional query optimizers to deal with UDFs. Query optimizers have largely been focused on optimizing declarative SQL queries. Such optimizers are fundamentally not designed for UDFs that might be called inside these SQL queries. They treat UDFs within a SQL query as a black box [40]. This leads to a lot of degradation in query performance and potential cost optimization opportunities.

For example, Listing 1.3 effectively produces the output of Listing 1.5. Listing 1.5 is fully declarative and will effectively take one pass over the `ORDERS`. On the other hand, an optimizer that treats `totalRevenue` as a black box will not optimize Listing 1.1 and effective produce a cursor loop over `CUSTOMER`. Each iteration of the cursor loop will involve the filter query over `ORDERS` in the body of `totalRevenue`.

```
SELECT c.C_CUSTKEY, SUM(O_TOTALPRICE) FROM CUSTOMER c LEFT OUTER JOIN
↪  ORDERS o ON c.C_CUSTKEY = o.O_CUSTKEY GROUP BY c.C_CUSTKEY;
```

Listing 1.4: Optimized equivalent of Listing 1.3

```
SELECT c.C_CUSTKEY, p.* FROM CUSTOMER c, LATERAL (SELECT
↪  SUM(O_TOTALPRICE) FROM ORDERS WHERE O_CUSTKEY = c.C_CUSTKEY) p;
```

Listing 1.5: Optimized equivalent of Listing 1.3

Because of such performance issues with UDFs, research until recently has been focused on moving analytical processing to higher-level frameworks such as Spark [14]. Some recent work looks into unifying the declarative and imperative paradigms to address such inefficiencies. One line of work has been tailored towards converting imperative UDFs into SQL that can be inlined into any calling query as a SQL subquery [42]. We further discuss this technique of query inlining in the next section.

1.2 Query Inlining

Query inlining is a technique that aims to address the declarative/imperative disconnect between SQL and procedural languages by converting imperative UDFs to declarative inlinable SQL. By converting functions to inlinable SQL, the DBMS can execute a combination of SQL and a UDF as pure SQL. This pure SQL now does not have any of the limitations discussed in Section 1.1. For example, a DBMS can take the body of `totalRevenue` from Listing 1.1 and inline it in its call-site as seen in Listing 1.3.

An important aspect of this inlining is the `LATERAL` keyword. This keyword appears in a `FROM` list. Subqueries on the right hand side of the `LATERAL` keyword can reference tables and subqueries on the left side of the keyword like in Listing 1.6. In other words, the subquery on the right side of `LATERAL` executes for each tuple output from the left side.

Inlining can leverage this behavior to emulate control flow between two successive imperative statements. The `LATERAL` keyword from PostgresSQL and the SQL Server equivalent, `APPLY`, have

```
SELECT * FROM table_1, LATERAL (SELECT y FROM table_2 WHERE table_1.id
↪  = table_2.id) p;
```

Listing 1.6: An example usage of `LATERAL`. The subquery on the right hand side of `LATERAL` can freely reference `FROM` entries to the left ot the keyword.

```
CREATE FUNCTION cursorExample () RETURNS DECIMAL AS
$$
DECLARE
    sum DECIMAL := 0;
BEGIN
    FOR P IN (SELECT O_TOTALPRICE FROM ORDERS) LOOP
        sum += P*P
    END LOOP;
    RETURN sum;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 1.7: Example of PL/PGSQL UDF that uses a cursor loop

been supported for the last decade [25, 7]. Inlining finds SQL equivalents for each UDF language construct. Ramachandra et al. made a framework called Froid that takes a subset of T-SQL constructs and converts them to inlinable SQL [42]. Froid has a multi-stage process of converting T-SQL to inlinable SQL. First, the function is divided into regions.There are two different types of regions: 1) sequential and 2) conditional. Conditional regions contain `if..else..` statements or loops while sequential ones do not. This classification helps with the next phase, that maps the individual regions to relational expressions. This is done on a case-by-case basis regarding the type of the imperative T-SQL statement that is being mapped. For example, **`IF..THEN..ELSE`** statements are mapped to **`SELECT CASE WHEN ... THEN ... ELSE ... END`** statements. Note that some imperative statement types, such as loops, do not have convenient relational statement mappings. These constructs are unsupported by Froid. These sets of relational regions are then chained together using the **`APPLY`** operator. After this, we expect to have one single relational SQL statement to represent the entire query. Now the optimizer can do single statement transformations to originally multiple T-SQL statements, resulting in optimizations equivalent to that of an imperative compiler. Note that this does not support recursive functions either as the resulting statement size would blow up endlessly.

Gupta et al. extended Froid to support cursor loops in an extension called Aggify [28]. Cursor loops iterate over the results of a query and execute a body of code for each tuple like in Listing 1.7. Aggify packs the logic of a cursor loop's body into a custom aggregate function and utilizes that aggregate instead as the output consumer to the cursor loop query. Listing 1.9 is an example of Aggify's operation on Listing 1.7.

Duta et al. extended these inlining techniques to support PL/PGSQL loops as well [21]. They used the **`WITH RECURSIVE`** operator to emulate loops. They implemented a framework called Apfel that takes PL/PGSQL loops and convert them to an inlinable equivalent that uses the PostgreSQL **`WITH RECURSIVE`** operator. Duta et al. also created a more optimized version of **`WITH RECURSIVE`** called **`WITH ITERATIVE`** to further optimize these loop transformations. Chapter 5 discusses this

```
SELECT cursorExample();
```

Listing 1.8: Invocation of Listing 1.7

```
SELECT customAgg(O_TOTALPRICE) FROM ORDERS;
```

Listing 1.9: Aggify's output from operating on Listing 1.7. `customAgg` is a dynamically created aggregate function that sums the squares of all elements given to it.

process in further detail.

Inlining frameworks are able to convert UDFs into a SQL subquery that an optimizer can transform. Ramachandra et al. noted that the transformations the SQL Server optimizer ends up doing on a given UDF is similar to the optimizations an imperative compiler would have done on the original UDF formulation. This observation shows an important benefit to inlining UDFs to SQL.

In this thesis, we compare the performance and supported UDF constructs of this inlining framework with another UDF execution framework based on query compilation. We discuss query compilation in the following section.

1.3 Query Compilation

DBMSs can be disk-based or they can be in-memory. Disk-based databases such as Oracle RDBMS, PostgresDB, SQL Server use disk as their main storage medium [12, 13, 17]. The disk stores tables and indexes in these DBMSs. On the other hand, in-memory DBMSs rely on main memory for storing their indexes and tables. SingleStoreDB, HyPer, NoisePage and VectorWise are examples of in-memory DBMSs [5, 1, 3]. While disk-based systems offer slower data access speeds than in-memory systems, they are cheaper as RAM space is more expensive than disk space.

In both types of systems, an incoming SQL query is parsed into a parse tree. A query optimizer eventually turns this parse tree into a query plan tree. This tree represents how the requested query will be executed. It consists of operator nodes that represent how a certain operation will be executed. Each node is abstracted to produce tuples for its parent nodes. For example, the query **`SELECT x FROM sample_table WHERE x=5;`** could be optimized to a tree consisting of a filter node that parents a sequential scan node. The filter node would contain the expression **`x=5`** that it evaluates. The query could also be optimized to almost the same tree with the scan node replaced with an index scan node if there is an index on `x`. After the optimizer produces this tree, the query execution engine of the DBMS executes the resulting query plan tree to complete the query.

Many disk-based DBMSs execute optimized plans in an Iterative model that is some variant of having every operator implement a **`next()`** function that returns the next tuple that the operator can output, modeled after iterators [27]. Expressions implement an **`evaluate()`** function that takes in input tuples and outputs the result of applying the input's underlying expression. Frequent virtual function calls incur CPU overhead in in-memory DBMSs. The materialization model avoids this overhead by having each operator generate all its output results in an in-memory buffer before proceeding to the next operator. However, this makes execution performance bound by memory accesses. All these overheads were suitable in disk-based DBMSs as the overhead of disk access overtook all of these pitfalls. This is not the case for in-memory DBMSs where the main bottlenecks are CPU execution and memory-accesses.

In order to sidestep such bottlenecks, the current norm for in-memory DBMS query engines is either vectorization or compilation [30]. Vectorization operates similar to the iterative model, except that each operator pulls a "vector" of tuples at a time, amortizing the cost of pulling. Processing on these tuples are done via specialized primitives that are designed to operate on batches of inputs. Compilation, on the other hand, operates by converting an entire query plan to an imperative intermediate representation (IR) such as C, C++, LLVM IR, or even a custom domain specific language and compiling that to machine code. This machine code can be repeatedly executed in order to complete the requested query. When executing this compiled code, the query execution engine does not have to deal with any overheads of interpreting the query operator nodes, such as virtual function calls, as it produced a hard-coded module whose execution models the given query plan. All the interpretation overheads are frontloaded and incurred during the compilation. We discuss further details of this compilation process in Chapter 3.

This compilation model converts all declarative SQL queries to an imperative destination language. We can compile imperative UDFs into a function in this same destination language. Compiled SQL queries that call this UDF can invoke a call to that function. The fact that we are compiling one imperative language to another simplifies this compilation process. For example. `WHILE` loops in PL/SQL can simply turn into `WHILE` loops in C. This technique of UDF execution is similar in a way to the inlining technique because both techniques aim to express UDFs and SQL queries in the same IR. Inlining achieves this goal by reexpressing UDFs in SQL while compilation does so by reexpressing UDFs and SQL to the same imperative destination language. However, inlining does this reexpression before query optimization takes place while compilation does so after.

1.4 Contribution

Previous works discuss executing UDFs via compilation or via inlining in depth. In this thesis, we aim to provide an experimental comparison of the two techniques. We do a survey of currently used UDFs from GitHub and analyze used language constructs in the context of comparing inlining and compilation support. We also implement infrastructure for both UDF compilation and inlining in a single main-memory DBMS to provide an apples-to-apples performance comparison. This process involves implementing a new UDF compilation framework.

The rest of this thesis is laid out as follows. Chapter 2 discusses an analysis of UDFs that we found on GitHub in order to motivate this comparison. We then discuss the implementation of our UDF compilation framework and infrastructure we implemented to support inlining in Chapter 3. We then compare the two UDF execution techniques in Chapter 4 before discussing related works in Chapter 5. We finally conclude in Chapter 7 followed by future works in Chapter 6.

Chapter 2

Survey of UDFs

To compare UDF language support across inlining and compilation, we first need to gather an understanding of the space of UDFs that currently exist. It is useful to know what kind of UDF constructs are in use and how many of these can or cannot be inlined by current query inlining methods. Such an analysis gives us a better understanding of the overlap between compilation and inlining in UDF language support. Furthermore, this analysis also provides a motivation on which language features with which we can compare inlining and compilation performance.

We used GitHub as a source to sample UDFs for our analysis under the assumption that these publicly available UDFs form an accurate sample of modern applications. As we implemented our UDF compilation framework to operate on PL/PGSQL UDFs, we analyzed such functions on GitHub. By making a web scraper, we extracted 4102 repositories to extract 19408 publicly indexed PL/PGSQL functions from GitHub in July 2020 [36]. We then processed these functions using the Postgres parser and created aggregate analytics on the resulting parse tokens. Each parse token corresponded directly to a PL/PGSQL construct.

Out of the all the functions we scraped, we found 12.2% to have non-inlinable features. The statistics of these non-inlinable features and other significant inlinable features are summarized in Table 2.1. We go through some of these constructs and consider why they can or cannot potentially be inlined in a SQL query. The results of the survey are summarized in Table 2.1.

2.1 SQL Execution

A SQL execution statement in PL/PGSQL entails a SQL statements that can populate a PL/PGSQL variable with the output its output. These SQL statements must be scalar: they return at most one tuple. If the query is not scalar, one of the tuples in the query output is populated into the variable. In Table 2.1, we distinguish between read-only SQL statements and DML changes (inserts, updates and deletes) as the two categories have different degrees of compatibility with inlining.

Feature	Count	Percentage (%)	Inlinable?
SELECT statements	8492	43.7	yes
IF statements	5893	30.4	yes
RAISE statements	1667	8.6	no
Cursor Loops	1588	8.2	yes
INSERT/DELETE/UPDATE statements	1391	7.2	no
FOR/WHILE loops	786	4.0	yes
Dynamic Execution	696	3.6	no
EXCEPTION blocks	571	2.9	no

Table 2.1: Results of Github Survey of PL/PGSQL Features.

```sql
CREATE FUNCTION sqlExecExample () RETURNS DECIMAL AS
$$
DECLARE
    result DECIMAL := 0;
BEGIN
    SELECT SUM(O_TOTALPRICE) INTO result FROM orders;
    RETURN result;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.1: Example of PL/PGSQL UDF that uses a SQL execution construct to populate the `result` variable.

```sql
CREATE FUNCTION insertExample (val INTEGER) RETURNS VOID AS
$$
BEGIN
    INSERT INTO sample_table VALUES (val);
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.2: Example of PL/PGSQL UDF that inserts into sample_table the parameter value, `val`.

```sql
SELECT insertExample(42);

/*

insertExample
--------

(1 row)
*/
```

Listing 2.3: Invocation and output of `insertExample` from **??**. It runs succesfully and returns NULL to the client.

`SELECT` statements are inlinable directly as a SQL subquery [42]. As we explain in Chapter 3, this is compilable as well. This construct is the most frequently appearing major construct amongst the functions that we scraped.

DML changes, however, are not as easily inlined. Suppose a SQL query equivalent for an insert existed in order to inline Listing 2.2. We would need the transformed equivalent of **`SELECT insertExample(42);`** to return a NULL tuple and also perform the insertion. To satisfy the latter, the transformed query would need to have an **`INSERT`** clause inside it. The first issue with this is only **`SELECT`** statements are allowed inside subqueries [15, 11]. Besides that, there are a few barriers to an **`INSERT`** clause being inside a subquery. As **`INSERT`** operators do not pass anything up its query plan the insertion must be at the root of the query plan. This contradicts the other requirement that a NULL tuple must be returned to the client. Therefore, there is no valid SQL that performs the equivalent of **`SELECT insertExample(42);`**. Without the capability to pass up information about the success of an **`INSERT`**, inlining exception-handled inserts is non-trivial. The same issue lies in other DML change query types as well. In order to sidestep this issue, we can follow the idea of Aggify and make a new operator to suit our needs. In this case, an operator that performs the effect of an **`INSERT`** but passes a NULL tuple to its parent would suffice. We elaborate on the complexity of adding such new operators at the end of this chapter.

```sql
CREATE FUNCTION absVal (test INTEGER) RETURNS INTEGER AS
$$
DECLARE
BEGIN
    IF test >= 0 THEN
        RETURN test;
    ELSE
        RETURN -1 * test;
    END IF;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.4: Example of PL/PGSQL UDF that uses a conditional construct to populate the absolute value of the input parameter.

Compilation does not face the same issues as it does not rely on the generated code to be a single relational SQL query. It can generate code for the insertion and then generate any logic after this code as well.

2.2 IF statements

IF statements in PL/PGSQL function the same way as any conventional imperative conditional block. Listing 2.4 has an example of this construct for reference. This example uses a conditional construct to populate the absolute value of the input parameter by casing on the sign of the input.

This construct is inlinable by converting this construct to a SQL **`CASE..WHEN..THEN..ELSE..END`** construct [42]. A compiler can simply transform such conditional blocks to a conditional jump in an imperative language.

2.3 Raise Statements and Exceptions

`RAISE` statements provide logging mechanisms and a way to raise exceptions for PL/PGSQL function writers. They take in an enum parameter that specifies what the severity of the logging message being raised is. This severity level can be `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, or `EXCEPTION` in increasing severity of the message. Parameters after the severity level describe the message by including a string with format specifiers and arguments to populate the format specifiers. If a `RAISE` statement with level `EXCEPTION` is executed, then it does the equivalent of throwing an exception in Java or C++. This exception needs to be caught by a matching PL/PGSQL `EXCEPTION` block lest it gets thrown as an error to the client, aborting the current transaction. Listing 2.5 provides a few examples of this construct.

```sql
CREATE OR REPLACE FUNCTION divide (a INTEGER, b INTEGER) RETURNS
↪  INTEGER AS
$$
DECLARE
BEGIN
    IF b = 0 THEN
        RAISE EXCEPTION 'INVALID DIVISION';
    ELSE
        RAISE NOTICE 'SUCCESSFUL DIVISION';
        RETURN a/b;
    END IF;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.5: An example of `RAISE` statement usage using a function that divides its two inputs. It raises an exception if the given denominator is zero.

```sql
CREATE OR REPLACE FUNCTION safedivide (a INTEGER, b INTEGER) RETURNS
↪  INTEGER AS
$$
DECLARE
res INTEGER := 0;
BEGIN
    res := divide(a, b);
EXCEPTION
    [when others then
    RAISE WARNING 'BAD DIVISION, RETURNING DEFAULT VALUE';
    RETURN -1;
    ]
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.6: An example of `RAISE` statement and `EXCEPTION` block usage using a function that wraps a call to `divide` from Listing 2.5 with an `EXCEPTION` block to catch any divisions by zero.

```sql
SELECT divide(6,3);

/*
NOTICE:  SUCCESSFUL DIVISION
divide
--------
     2
*/

SELECT divide(5,0);

/*
ERROR:  INVALID DIVISION
CONTEXT:  PL/pgSQL function divide(integer,integer) line 5 at RAISE
*/
```

Listing 2.7: Example invocation and output of `divide` from Listing 2.5

```sql
SELECT safedivide(5,0);

/*
WARNING:  BAD DIVISION, RETURNING DEFAULT VALUE
divide
--------
     -1
*/
```

Listing 2.8: Example invocation and output of `safedivide` from Listing 2.6

These `RAISE` statements are present in a plurality of non-inlinable functions. This makes sense as they are useful for debugging.

An interesting aspect about these statements is that the string they output may not be of the same schema of the overall query that the client requested. For example in Listing 2.7, the schema of each call to `divide` is one integer. However, as seen in the function invocations in Listing 2.7 and Listing 2.8, the `RAISE` messages print strings to the client that do not match this single integer schema. In the context of inlining, this mismatch poses a difficulty as SQL operators in a query plan are known to pass up tuples to fit the schema that is expected of them. In effect, no current SQL operator can correctly emulate the effect of a `RAISE` statement and support its inlining. In fact, there is no SQL operator that can print anything to the client that is not part of the output query result. One would have to make a brand new SQL operator that adds a message to a buffer that is eventually flushed to the client. Such a SQL operator would have to be able to emulate the exception raising capabilities as well.

Compiling a `RAISE` statement would involve generating code that adds a message to a buffer that is eventually flushed to the client. This generated code matches the capabilities of the hypothetical `RAISE` SQL operator discussed in the previous paragraph. If the `RAISE` is an exception statement, a UDF compiler can generate the equivalent of a `throw` statement in Java or C++, or whatever the equivalent is in the destination language. This compilation is simplified by the fact that most imperative languages have some sort of exception handling pattern that programmers use. Compiling a `RAISE EXCEPTION` statement involves producing one of these patterns.

Exception blocks pair with `RAISE EXCEPTION` constructs and to catch other sources of exceptions such as a division by zero. Inlining these constructs could involve transforming these exception blocks to conditional blocks based on error condition codes. These error condition codes would have to be set by any potentially hazardous operation such as division or modulo. After a rewriter converts these exception blocks to conditional statements, it can inline the resulting conditional statements using Froid [42]. For example, **`EXCEPTION`** It blocks were mostly used to detect the success of a DML change such as an **`INSERT`** like in Listing 2.9.

It is non-trivial to find an inlinable SQL equivalent for the **`EXCEPTION`** block in Listing 2.9. As discussed in Section 2.1, inlining an **`INSERT`** requires writing an additional operator. This operator can pass up error or success information via an error code integer and have the parent case on it.

In effect, the capabilities inlining needs to catch DML failures would require modifications to the SQL standard as well or implementing some custom operator. For the same reason, inlining **`INSERT`** statements in a SQL query would also be non-trivial.

On the other hand, compiling exceptions involves producing the equivalent of the exception handling pattern in the destination imperative language. In Java and C++ , this pattern can map to `catch` blocks. In C this can map to `setjmp/longjmp` constructs. Compilation can handle Listing 2.9 more efficiently. A compiler's generated code for an **`INSERT`** could throw an exception on failure. A generated exception block can catch this exception.

```
CREATE OR REPLACE FUNCTION safeinsert (a INTEGER, b INTEGER)
RETURNS INTEGER AS
$$
DECLARE
    res INTEGER := 0;
BEGIN
    INSERT INTO sample_table VALUES (a,b);
EXCEPTION
    [when others then
    RAISE WARNING 'Failed Insertion';
    RETURN -1;
    ]
RETURN 1;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.9: Example of a function that handles a failure to **`INSERT`** into **`sample_table`**. It returns -1 on failure and 1 on success.

2.4 Loops

We have two broad categories of loops in our survey: 1) cursor loops and 2) non-cursor loops. Non-cursor loops are for/while loops that iteratively increment a variable and stop on a condition. These are important constructs that are used in every imperative language. Compiling these non-cursor loops involve producing their equivalents in the destination language. For example, WHILE loops in PL/PGSQL can be compiled directly to a while loop in C++. The body of the produced C++ loop is the compiled body of the PL/PGSQL loop. On the other hand, Duta et al. has shown that these loops can be converted to inlinable SQL as well [21] using **`RECURSIVE CTE`** constructs as discussed in Section 1.2.

Cursor loops iterate over the result of a query. Cursor loop inlining is discussed in Section 1.2 and the Aggify framework can do this inlining [28]. Much like the hypothetical `RAISE` operator discussed in Section 2.3, the Aggify framework creates a new custom aggregate operator for each cursor loop. They both involve making new operators to support inlining `RAISE` statements and cursor loops.

Compiling cursor loops is not as simple as writing a for loop. The complexity of this compilation depends on the method by which a query engine compiles a SQL query. We further discuss details on this compilation in Chapter 3.

```
CREATE FUNCTION registerNewCustomer(new_cust_id INTEGER) RETURNS VOID
 ↪  AS
$$
DECLARE
    currentMonth INTEGER := NULL;
BEGIN
    .
    /* code that populates the current month in currentMonth */
    .
    .
    EXECUTE 'INSERT INTO NewCustomer_' || 'currentMonth' || ' VALUES
     ↪  ($1)' USING new_cust_id;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 2.10: Example of dynamic SQL **EXECUTE** usage. This function inserts the id of a new customer into a specific table. This table depends on what month it currently is. Note that the '||' is a string concatenation operator in PL/PGSQL.

2.5 Dynamic Execution

A significant source of non-inlinable features in Table 2.1 is dynamically executed functions. These are statements that invoke a SQL query whose definition is decided at runtime through a string template with format specifiers along with variables to populate the placeholders with. The strings to these dynamically executed statements involve consistent patterns, with a frequent one being where we insert values into a table whose identity decided at runtime like in Listing 2.10.

The fact that the executed SQL query in these **EXECUTE** statements is decided at runtime makes inlining these constructs difficult. **??** itself poses a challenge for inlining. Apart from the fact that DML change statements can't trivially be SQL-inlined, queries with a variable table name can only be inlined if all the possible table names this variable can represent are known at the time of UDF compilation. In this case, all the possible table names are **NewCustomer_1**, **NewCustomer_2**, **NewCustomer_3**...**NewCustomer_12**. Suppose an inlining framework knew about this limited space of possible tables, it would then have to generate SQL for each table case. In effect, attempting to inline such constructs by enumerating possible SQL queries results in a query size explosion.

Interestingly, many of the aforementioned PL/PGSQL constructs are non-inlinable with the current SQL standard but can be made inlinable by extending the SQL interface. This common theme emphasizes that the space of SQL-inlinable functions is limited by the space of possible combined functionalities of existing SQL operators. This is a limitation as these operators were not necessarily created with imperative computation as a considered use case. Because of this limitation, extending the space of SQL-inlinable functions maps to extending the space of existing SQL operators. Aggify's creation of custom aggregate operators for cursor loops exemplify such an extension [28]. The introduction of new SQL operators can come with significant development overhead. On top of that, the optimizer of the concerned DBMS must also be able to produce efficient plans with these new operators. The main motivation of SQL inlining is better optimization of an imperative query. A SQL optimizer performs transformations on the inlined query that are analogous to whatever optimizations an imperative optimizer would produce on the original UDF [42]. Introducing new operators would mean introducing new optimizer functionality to maintain this property, and potentially added complexity to the optimizer.

This analysis motivates exploring the effectiveness of compiling UDFs to an imperative IR over a SQL-inlinable IR. This investigation is especially worthwhile in a DBMS that already converts all incoming queries into imperative LLVM IR. This thesis limits the scope of this investigation to implementing a UDF compiler that can match the capabilities of Apfel and Froid, benchmark them against each other and analyze high-level sources of any performance discrepancies amongst the two UDF compilation paradigms.

Chapter 3

UDF Compilation Architecture

The current inlining capabilities we discussed in Chapter 2 mainly involved SQL execution, for loops, while loops and cursor loops. In order to have an apples-to-apples comparison, we implement compilation support for these constructs in a single system. We implement these features in NoisePage, a main-memory DBMS that executes queries via compilation [3].

In this chapter, we examine the current query compilation architecture of NoisePage. Next, we discuss how we implemented our UDF compilation framework on top of that. We finish off by discussing what features we added in NoisePage to support inlining.

3.1 NoisePage Query Compilation

The compilation architecture in NoisePage is based on the Data-Centric Compilation framework proposed by Neumann [37]. This framework breaks up a query plan from an optimized SQL query into pipelines. A pipeline represents an ordered series of relational operators that do not have to materialize its results before reaching the last operator in this series. It contains a series of operator translators, each of which represents a relational operator. Each translator object produces logic for one specific relational operator.

A pipeline breaker refers to the end of these pipelines where tuples must be materialized. For example, the build side of a hash join is a pipeline breaker as all tuples must be materialized into the hash table before proceeding. The build side of a nested loop join, however, does not demarcate a pipeline breaker. Furthermore, instead of maintaining a scheme where each operator in a pipeline pulls tuples from its children as in an iterative model, the data-centric framework does it the other way around. Operators push their tuples to their parents instead of having their tuples pulled by a function call from the parents. This is implemented by nesting the logic of an operator in that of its parent. This way, a tuple being passed upwards remains in a machine register for the length of the pipeline. This is not possible with statically generated code for each operator. Instead, a function is dynamically generated for each pipeline representing the logic for all the operators in this pipeline. NoisePage implements this by compiling each operator pipeline to a function in a domain-specific language called TPL. All functions for a query form a TPL

```sql
SELECT orders.o_custkey + 1 FROM orders
JOIN customer ON customer.c_custkey = orders.o_custkey
WHERE orders.o_orderkey < 155;
```

Listing 3.1: Join query on `ORDERS` and `CUSTOMER` with a filter and projection

module that is further compiled into bytecode called TBC. This bytecode can then either be run via interpretation or it can be further compiled to an LLVM module.

For example, NoisePage generates the query plan in Figure 3.1 for the query in Listing 3.1. We see that the query plan contains two pipelines. Each pipeline generates a separate function as seen in Listing 3.2. The TPL code in Listing 3.2 is then compiled to interpretable bytecode and then compilable LLVM IR. In order to execute Listing 3.1, we have the functions `pipeline1` and then `pipeline2`. NoisePage appends an output operator translator at the root of the plan in Figure 3.1 to generate code to serialize query outputs to a network buffer. This observation is important as we later change the contents of this output translator to compile cursor loops and other PL/PGSQL constructs.

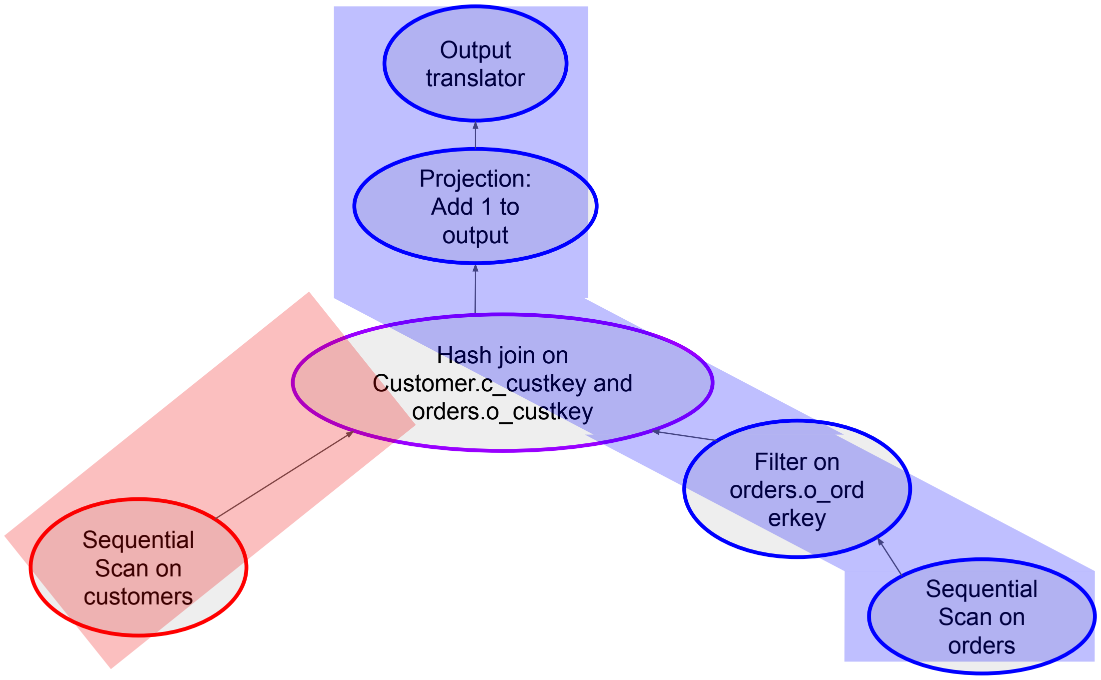

Output translator
Projection: Add 1 to output
Hash join on Customer.c_custkey and orders.o_custkey
Filter on orders.o_orderkey
Sequential Scan on customers
Sequential Scan on orders

Figure 3.1: Query plan of Listing 3.1. This plan is also divided into two pipelines. Operators belonging to pipeline 1 are shaded red while those in pipeline 2 are shaded blue. There is also an output translator at the root of the tree to generate code for sending query results to a network client.

```
fun pipeline1(QueryState *q) {
    TableIterator tvi;
    @tableIteratorSetup(&tvi, customers_oid)
    // sequential scan on customers
    for(@tableIteratorAdvance(&tvi)){
        // hash join table insert
        @hashTableInsert(q.join_ht, @getTupleValue(&tvi, 3))
    }
}

fun pipeline2(QueryState *q) {
    TableIterator tvi;
    @tableIteratorSetup(&tvi, orders_oid)
    // sequential scan on orders
    for(@tableIteratorAdvance(&tvi)){
        var o_custkey = @getTupleValue(&tvi, 1)
        // filter
        if(o_custkey < 155) {
            // hash join table lookup
            if(@hashTableKeyExists(q.join_ht, o_custkey)){
                // write to output client
                var out = @outputBufferAlloc(q.output_buff)
                // add 1 to output
                out.col1 = o_custkey + 1
            }
        }
    }
}
```

Listing 3.2: Translation of Figure 3.1 into TPL. Each pipeline has its own function. Executing Listing 3.1 amounts to executing functions `pipeline1` and `pipeline2`.

```
1  CREATE FUNCTION totalRevenue (highestKey INTEGER) RETURNS DECIMAL AS
2  $$
3  DECLARE
4      sum DECIMAL := 0;
5  BEGIN
6      FOR order_price IN (SELECT O_TOTALPRICE FROM ORDERS WHERE
        ↪  O_ORDERKEY < highestKey) LOOP
7          sum += order_price
8      END LOOP;
9      RETURN sum;
10 END;
11 $$ LANGUAGE PLPGSQL;
```

Listing 3.3: Example of function with a cursor loop

3.2 UDF Compilation

We now discuss the UDF compiler that we implement on top of NoisePage's query compilation engine.

3.2.1 Cursor Loops and SQL Queries

In NoisePage, we implemented code generation for PL/PGSQL UDFs. We create a TPL module that represents the execution of a requested PL/SQL function. When the DBMS server receives a `CREATE FUNCTION` command, it uses the existing code generation architecture to convert the PL/PGSQL function to a TPL abstract syntax tree. Our framework does this conversion by interpreting the PL/PGSQL statements line-by-line and adding statements to the TPL function being built to correspond to each line. This works because both the source PL/PGSQL code and TPL follow an imperative paradigm. This imperative matchup between the two languages falls apart when processing a SQL statement or a cursor loop within the PL/PGSQL.

When compiling a SQL query like line 6 of Listing 3.3, the compilation framework can leverages NoisePage's code generation mechanism for converting SQL queries to TPL code. Note that `totalRevenue`'s cursor loop query captures its output into the variable **`order_price`**. Suppose the statement involved no such capturing and only the SQL query's execution was needed. In that case, compiling this line reduces to compiling standard SQL queries as in Section 3.1 and generating function calls to the pipelines in the resulting module. In this case, however, the generated code must capture a reference to the **`order_price`** variable and allow the compiled query to populate this reference with its output. Furthermore, the body of the cursor loop needs to execute for each tuple resulting from the query. Instead of having the query serialize its output to a network buffer as depicted in Listing 3.1, we must have the query serialize its output to

```
var y = 11
var myFunc = lambda [y] (z: Integer ) -> nil {
                    y = y + z
                }
myFunc(10)
```

Listing 3.4: Lambda function example in TPL. The variable `y` is captured by `myFunc`. `myFunc` takes in an integer input and does not output anything. It adds the input to the captured `y` variable. An invocation of `myFunc` follows.

`order_price` and execute the cursor loop body. In order to do this, the TPL language needs to have a data type that can

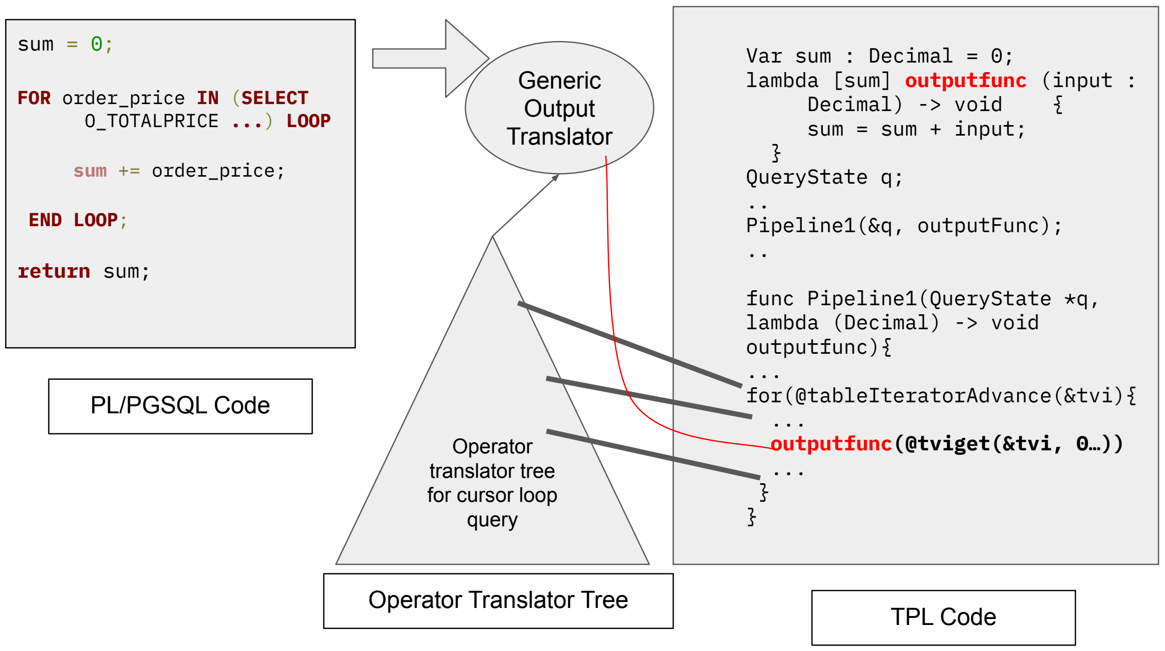
1. capture writeable references to local variables in the scope of the type's construction
2. specify parameterized logic using these captured variables

These two requirements are actually satisfied by lambda functions in C++. Taking inspiration from that, the UDF compilation framework also introduces lambda functions to TPL. Introducing this construct to satisfy the two requirements was a key component to supporting cursor loop compilation. The TPL syntax in Listing 3.4 corresponds to a lambda function.We discuss details of implementing lambda functions in NoisePage later in Section 3.2.2.

Using this lambda construct, the compilation framework transforms cursor loops by making a more generic version of the output translator depicted in Figure 3.1. Instead of creating a static output translator that only generates code to serialize into the network buffer, the framework's generic output translator takes in a lambda function that describes what logic to perform on each output tuple. In the case of cursor loops, a lambda function that is passed down captures all PL/PGSQL variables in the current function and performs compiled logic of the loop body.

We illustrate this process in Figure 3.2. Here, the UDF compiler generates the operator translator tree for the cursor loop query. The compiler appends a generic output translator at the root of the translator tree. This generic output translator declares a lambda function called `outputfunc` whose body performs the logic of the cursor loops body by accumulating the given input into the `sum` variable. This input is expected to be an output from the cursor loop query. Note that the lambda captures a writeable reference `sum` variable. The generic output translator generates a call to `outputfunc` for each output tuple. This way the lambda body is executed for each output tuple, executing the cursor loop.

This idea of packing the cursor loop body into a lambda function is similar in spirit to Gupta et al.'s approach with Aggify [28]. Gupta et al. similarly put the logic of a cursor loop's body into a custom aggregate function and utilizes that aggregate instead as the output consumer to the cursor loop query. The lambda functions in our approach form a lightweight counterpart to Aggify's custom aggregates. A shortcoming of our approach relative to Aggify is that Aggify allows the cursor loop body to execute with the same degree of parallelism as aggregates. In our approach, the lambda function executes iteratively on each output tuple. The application of the lambda function can potentially be vectorized but we leave that as future work.


Generic Output Translator
Operator translator tree for cursor loop query
PL/PGSQL Code
Operator Translator Tree
TPL Code

```
sum = 0;

FOR order_price IN (SELECT
      O_TOTALPRICE ...) LOOP

     sum += order_price;

 END LOOP;

return sum;
```

```
Var sum : Decimal = 0;
lambda [sum] outputfunc (input :
     Decimal) -> void    {
     sum = sum + input;
  }
QueryState q;
..
Pipeline1(&q, outputFunc);
..

func Pipeline1(QueryState *q,
lambda (Decimal) -> void
outputfunc){
...
for(@tableIteratorAdvance(&tvi){
  ...
  outputfunc(@tviget(&tvi, 0…))
  ...
 }
}
```

Figure 3.2: Compilation of the cursor loop in Listing 3.3. The network buffer code generated by the output translator (see Figure 3.1) is replaced by a call to the lambda function `outputfunc`.

Another aspect of SQL queries within PL/PGSQL functions to note is that they may populate local variables with their outputs and use local variables in a read-only manner within the query. This is exemplified in Listing 3.3 where **`highestKey`** is used in the SELECT query's predicate function. The compiler deals with this by compiliing the query as a prepared statement with **`highestKey`** as a statement parameter. Pointers to the parameter(s) are passed to the query through the execution context.

The compilation framework transforms SQL execution statements in a way similar to cursor loops. Consider Listing 3.5 that finds the most expensive order in the **`ORDERS`** table whose id is less than the supplied parameter. Compiling the SQL statement and capturing its output into `priciest` reduces to compiling a cursor loop on this query whose body only consists of an assignment to `priciest`.

3.2.2 Lambda Functions

We next discuss the framework's implementation of TPL lambda functions. TPL lambda functions capture references to the specified variables into a new stack-allocated structure. In Listing 3.4, the **`myFunc`** variable holds a pointer to the aforementioned stack-allocated structure of captured

```
CREATE FUNCTION mostExpensive (highestKey INTEGER) RETURNS DECIMAL AS
$$
DECLARE
    priciest DECIMAL := 0;
BEGIN
    SELECT O_TOTALPRICE INTO priciest FROM ORDERS WHERE O_ORDERKEY <
     ↪  highestKey ORDER BY O_TOTALPRICE DESC LIMIT 1;
    RETURN priciest;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 3.5: Example of simple function with a SQL statement. This function returns the most expensive order whose key is below `highestKey`.

locals. After this, any line can directly treat **`myFunc`** like a standard function and call it with that syntax. When it is called, an extra parameter containing the pointer to the capture structure is passed into **`myFunc`**. In other words, calling **`myFunc`** extracts the pointer value stored in **`myFunc`** and passed this in as an extra parameter to any call to **`myFunc`**. However, we cannot alias this to a variable that is named something other than the original variable alias for the lambda expression, which in this case is **`myFunc`**. We made this decision to keep in line with TPL's original runtime architecture that did not provide support for function pointers as it only provided specialized bytecodes for each function. Furthermore, we decided to keep it that way to enable LLVM to perform global optimization at the lambda's call sites, such as inlining the lambda function's body.

During the bytecode generation phase of TPL compilation, a lambda expression spawns the allocation and generation of a TBC function for the lambda body. Any caller of the lambda function can generate a jump instruction into the function with a statically determined jump offset.

3.2.3 Calling a UDF

We now discuss how calling a UDF from a SQL query is implemented. At a high level, the UDF TPL is inlined along with its helper structures and functions into the TPL of the calling query. However, we cannot merely have the TPL abstract syntax nodes refer directly to the UDFs original AST. This is because the AST of the UDF and the calling SQL query are created in two different AST contexts. These contexts store necessary metadata, such as cached representations for built-in integer types. In effect, different context instances have different objects to represent the same built-in integer type. We wanted to avoid confusion resulting from these different representations for the same types and other builtin identifiers.

To address this, our compilation framework clones the UDF AST nodes into the context of a calling query. An optimization we can do is precompile this into a dynamically loaded

LLVM function. Such an optimization comes with its costs. Inlining the AST nodes before LLVM compilation allows the LLVM framework to optimize across the UDF and its calling query, potentially inlining. Precompiling the function into LLVM would make such interprocedural analysis non-trivial.

In this chapter, we described our UDF compiler framework's implementation in the NoisePage DBMS. By introducing lambda functions in TPL, the framework supports cursor loops. Apart from cursor loops, the framework supports most of the language constructs described in Chapter 2. It currently does not support dynamic execution, `RAISE` or `EXCEPTION` blocks. We believe that this framework's support can be extended for these constructs as future work. Nevertheless, the framework's language support matches that of Froid and Apfel, allowing us to compare the frameworks on the basis of query execution performance.

Chapter 4

Experimental Evaluation

We now evaluate our imperative UDF compiler. The experiments aim to compare the performance of code generated by UDF compilation with that of SQL-inlined code. Therefore, we limit the workloads to be PL/PGSQL functions that are supported both by inlining and our compilation framework. We run four different experiments for this comparison. The first experiment tests a lightweight workload that is meant to compare the base overheads of inlining and compiling a small function. All the experiments lay down a comparison on highly iterative functions. One of these iterative functions performs only iterative computation and no SQL queries. The others do various analytical loops on randomly generated data.

Chapter 1 discussed how SQL inlined queries depend on lateral joins. However, the code generation layer of NoisePage currently does not support these constructs. Because of this, we executed the inlined queries by passing them through the Postgres 12.3 optimizer, inferring the optimized plan from this output, and manually hand-compiling this plan to TPL.

All given execution timings represent the runtime of the compiled module for each query. It does not take into account the timing of any layer above the execution engine such as the optimizer or the parser. We produced these timings by inserting instrumentation code in the execution engine of NoisePage. Each timing data point was taken to be the average timing across five different runs.

We ran all experiments on an Ubuntu 20.04 machine with an Intel i7-101710 1.10GHz processor with 6 cores, 12 threads and 64GB RAM. All experiments materialize query results in memory and output them to a file in human-readable format, thereby removing any network transfer effects.

4.1 Overhead of Function Call

Our first experiment measures the overhead of calling a UDF in NoisePage. We compare using a simple function that just adds one to a given integer shown in Listing 4.1. Inlining transforms the query **`SELECT addOne(x) FROM table_1;`** to **`SELECT x + 1 FROM table_1;`** where `TABLE_1` is a table consisting of one integer column **`x`**. We also compared with **`SELECT x;`** to compare the

```sql
CREATE OR REPLACE FUNCTION addOne(x integer) RETURNS integer AS $$
DECLARE
BEGIN
    RETURN x+1;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 4.1: addOne Function

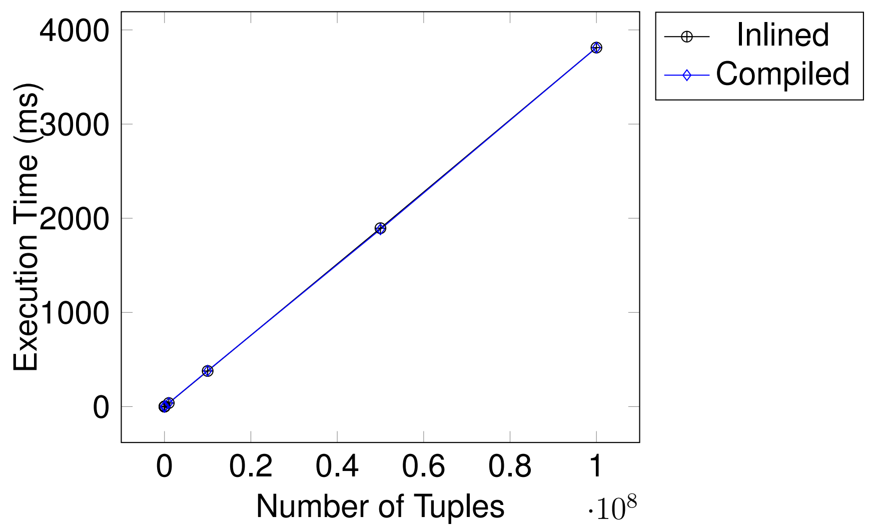


Figure 4.1: Results for addOne Benchmark

extent to which the inlined and compiled queries diverged in performance from this inlined query. This comparison would indicate any inefficiencies in the addition compilation. We evaluated these queries on a table with one integer column. We populated the column with randomly generated integers. For different data points, we increased the number of random tuples in the table up to 1 billion tuples and compared the query runtimes in milliseconds.

We see in Figure 4.1 that compilation and Inlining do not have any noticeable difference. In this plot, we see that the performance of **`SELECT x`** diverges from that the inlined and compiled queries. The inlined and compiled queries do not have any noticeable performance divergence. This lack of difference amongst the two suggests that the overhead introduced by the function call ranges from negligibly low to nonexistent. Upon inspection of the LLVM IR for the compiled query, we find that the function call was missing and had actually been inlined. The only runtime overheads of function inlining here included some callee-saved variables that were saved on the stack before entering the inlined function's context.

```
CREATE OR REPLACE FUNCTION sumNaturals(x integer) RETURNS integer AS $$
DECLARE
  ctr      integer := 0;
  result integer := 0;
BEGIN
  WHILE s <= x LOOP
    result := result + s;
    ctr      := ctr + 1;
  END LOOP;
  RETURN result;
END;
$$ LANGUAGE PLPGSQL;
```

Listing 4.2: `sumNaturals` function. This function sums the first x natural numbers and returns the results.

4.2 Lightweight Loops

The next experiment evaluates a UDFs performance on a tight loop with many iterations with the function in Listing 4.2. This UDF has no SQL queries inside it. Compilation of this is compared against the SQL inlinable version of this query the way Duta et al. transform the query using the Apfel system. We also make sure to use the more efficient **`WITH ITERATIVE`** construct that Duta et al. proposed instead of **`WITH RECURSIVE`** in the inlined queries [21]. Section 5.1 discusses the difference between the two constructs in further detail. This comparison aims to show how well our framework performs in UDFs with purely imperative constructs relative to SQL-inlining, which converts these imperative constructs to declarative syntax before evaluation.

To produce Figure 4.2 we ran `sumNaturals` from Listing 4.2 over the same single integer column table described in Section 4.2. In Figure 4.2(a), we see that the Apfel query performance linearly diverges from that of the compiled query significantly after the number of iterations exceeds 10^7. Before this point on the x-axis, the two graphs seem similarly poised in Figure 4.2. The scale of the y-axis hides the differences between the graph that are there. In that data range, compilation still outperforms the SQL inlined query by at least 10x on average as seen in Figure 4.2(b). The other data points at the higher iteration counts show a performance improvement in the compiled query by at least 120x.

This performance difference is because Apfel's inlined SQL query iteratively uses a temporary CTE table. Each iteration of the loop involves a scan and an insert into this CTE table. This process is rather inefficient as this particular use case of CTE tables comes with overheads that are not amortized by our usage pattern.

In applications that only involve SQL queries, the overheads of setting up a CTE table are amortized under the assumption that each scan on the table will read many tuples of data. Furthermore,

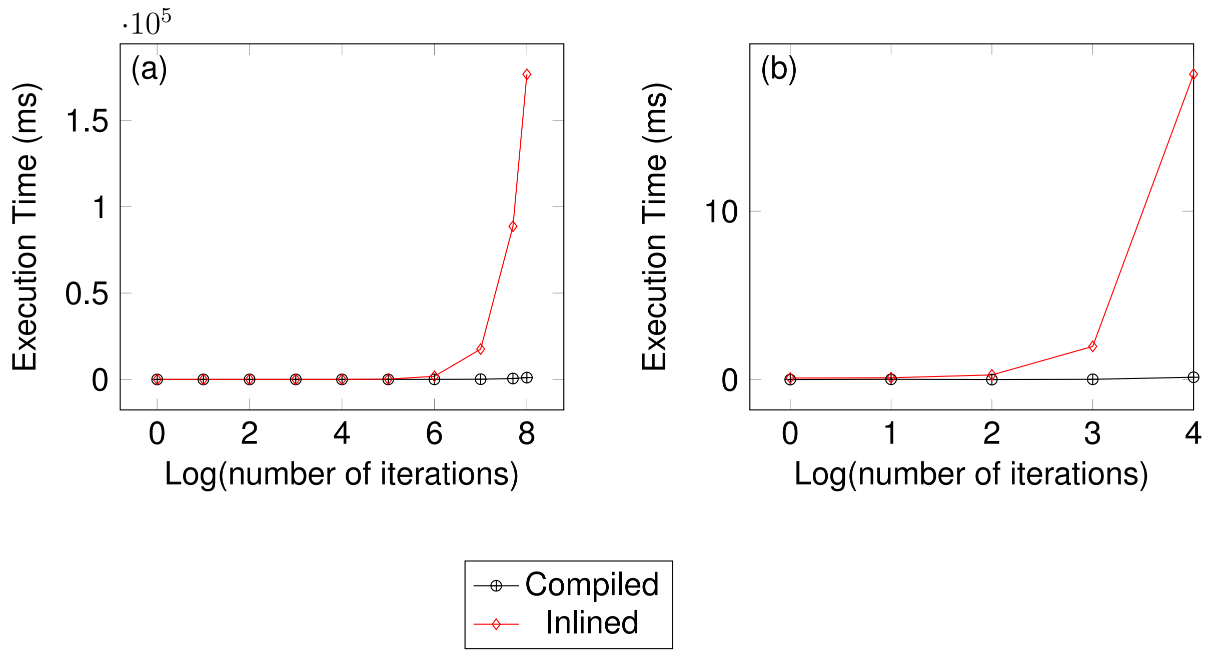


Figure 4.2: Results for sumNaturals Benchmark. (a) and (b) represent the same experiment except (b) has its domain's upper bound restricted to 10^4.

internal fragmentation in these tables would be minimized under the assumption that they hold many tuples. Neither of those assumptions are true in this use case of CTEs. Here, the temporary CTE table is effectively being used as a register for all the variables of one iteration of the loop through one tuple. Each iteration of the loop involves a SQL scan that invokes initialization overheads. We produced Figure 4.2 after minimizing these overheads by lowering the initial scan buffer allocation size to its minimum. The compiled version has neither of these overheads and simply does the required arithmetic operations directly to the SQL integer values represented as structures stored in memory. We see a performance gap between the two UDF execution frameworks, that only becomes wider because of this difference, as the number of loop iterations increases. Because Apfel transforms the UDF at the SQL query level, the SQL constructs it has to maintain at this level incur overheads that are not necessary at the TPL level. NoisePage's compilation at the TPL level bypasses these overheads. There are efforts to make iterative CTE execution more efficient, but each of them contain overheads that are larger than a compiled PL/PGSQL loop [24, 26].

4.3 L2Norm Benchmark

In this next experiment, we evaluate the performance of an algorithm that one may consider to be much better suited to run in a standard imperative runtime such as C as opposed to part of a procedural UDF such as T-SQL or PL/PGSQL. To do this, we devised a function with a double-nested `WHILE` loop with SQL queries inside.

```
CREATE OR REPLACE FUNCTION L2Norm(n integer) RETURNS integer AS
$$
DECLARE
  sum DECIMAL := NULL;
  i integer := NULL;
  j integer := NULL;
  a integer := NULL;
  b integer := NULL;
  row integer := NULL;
BEGIN
  sum := 0;
  i := 1;
  WHILE i <= n LOOP
    row := 0;
    j := 1;
        WHILE j <= n LOOP
            /* Compute dot product of ith row of table_1 and jth column
            ↪ of table_2 and add that to sum */

            FOR a,b in (SELECT p.v,q.v  FROM matrix_1 p JOIN matrix_2 q
            ↪ ON p.c = q.r WHERE p.r = i and q.c = j) LOOP
        row := row + a*b;
      END LOOP;
      j := j + 1;
    END LOOP;
    sum := sum + row * row;
    i := i + 1;
  END LOOP;
  RETURN sum;
END;
$$
LANGUAGE PLPGSQL;
```

Listing 4.3: L2Norm Function

```
CREATE OR REPLACE FUNCTION L2NormNoCursor(n integer) RETURNS integer AS
$$
DECLARE
    sum integer := NULL;
    i integer := NULL;
    j integer := NULL;
    a integer := NULL;
    b integer := NULL;
    elem integer := NULL;
    row integer := NULL;
BEGIN
    sum := 0;
    i := 1;
    WHILE i <= n LOOP
        row := 0;
        j := 1;
        WHILE j <= n LOOP
            /* Compute dot product of ith row of table_1 and jth column
             ↪ of table_2 and add that to sum */

            SELECT SUM(p.v*q.v) into elem FROM matrix_1 p JOIN matrix_2
             ↪ q ON p.c = q.r WHERE p.r = i and q.c = j;
            -- RAISE NOTICE '% , % , %', i,j,elem;
            IF elem IS NULL THEN
                elem := 0;
            END IF;
            row := row + elem;
            j := j + 1;
        END LOOP;
        sum := sum + row * row;
        i := i + 1;
    END LOOP;
    RETURN sum;
END;
$$
LANGUAGE PLPGSQL;
```

Listing 4.4: L2Norm Function from Listing 4.3 with the cursor loop removed

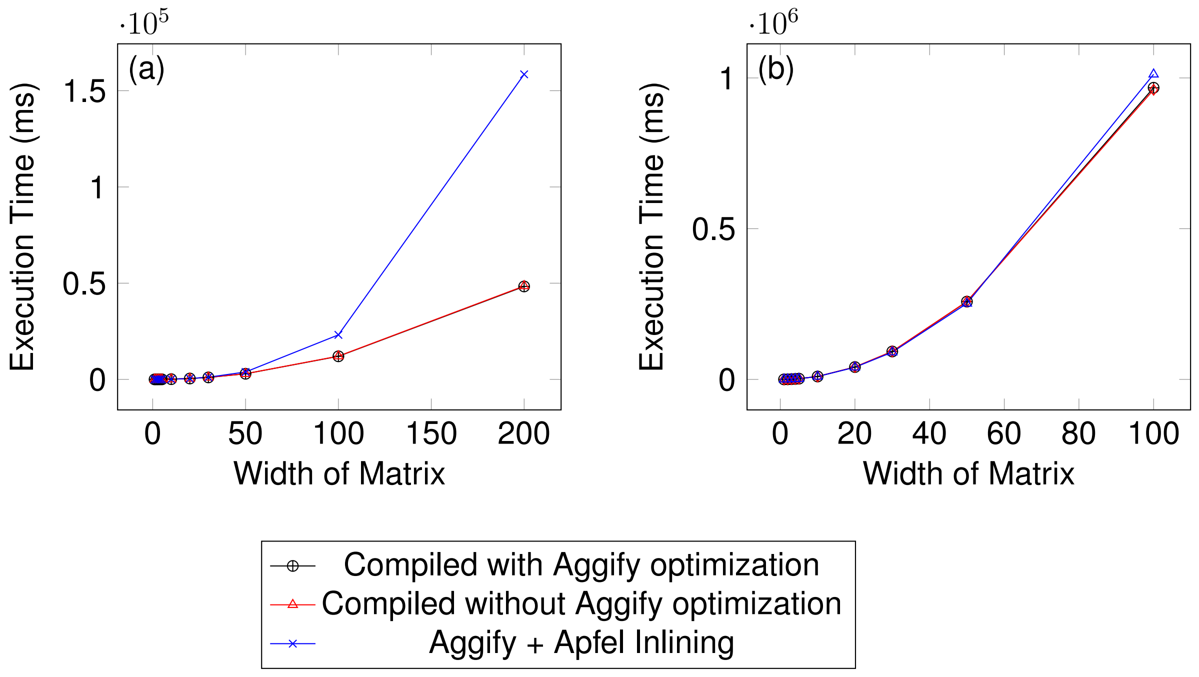


Figure 4.3: L2Norm Benchmark plots (a) with indexes and (b) without indexes. Note that the red and black plots overlap with each other in the lower left of both plots.

We created a PL/PGSQL function that effectively finds the Frobenius or L2 norm of a product of two matrices. We represent each matrix via a SQL table. There are many possible implementations of matrices in SQL, such as representing all matrices in one table, but we chose this for simplicity. The SQL table contains tuples of the form (row, column, value), each encoding a matrix element. We load 1 million tuples into each matrix table, representing a 1000 by 1000 matrix each. We then varied the input parameter, **n**, to the L2Norm function in Listing 4.3. The L2Norm UDFs parameter limits computation scope by only having it operate on the first **n** rows and first **n** columns of each matrix.

We also evaluated a minor rewrite of this function in Listing 4.4 with the Aggify paper's transformation to eliminate the cursor loop. Apfel inlines this rewritten function more efficiently. Apfel's performance on the rewritten query with the cursor loop removed is much better, so we decide to compare against this. This experiment produced three different sets of data points in Figure 4.3(a). The orange line graph plots the performance of running the original UDF compiled on NoisePage. The black line charts the performance of the "Aggified" UDF compiled on NoisePage. Lastly, the blue line plots the Aggified query's Apfel transformation.

We evaluated two different database configurations for this experimental: with and without indexes over the table's matrix rows and columns. Figure 4.3(b)does not involve such indexes while Figure 4.3(a)does. The loop's body performs much heavier computation when indexes are absent and thus presents much more of a bottleneck than in the setup with the indexes.

We see that in both plots of Figure 4.3, the black and red data points overlap and have no significant difference. This indicates that there is no significant performance benefits from Aggify rewrites in our compilation framework. As noted in Chapter 3, the implementation of cursor loops involves packing the loop body into a lambda function and executing that for each tuple in the cursor loop query. This is similar to making a custom aggregate and passing each tuple through that as in Aggify. That would explain why there is not much of a performance difference between the the compiled plots with and without Aggify applied.

In Figure 4.3(b), we see that the SQL-inlined queries start to diverge in performance relative to the compiled queries after the width of the multiplied matrices is greater than 100. This discrepancy is because of the difference between the TPL IR produced by both these query types. The compiled queries perform PL/PGSQL loop iteration by incrementing a variable and changing the program counter. In contrast, the Apfel query requires an entire scan and insertion onto a temporary CTE table. Again, the differences between these two are more pronounced as the number of loop iterations increases. Figure 4.3(a)further backs up this claim by showing that if we make the body of the loop computation lighter, then we observe an even larger divergence in performance. In this case, any overhead caused by loop iteration contributes a larger percentage to the computation since overheads in other parts of the loop have reduced. Thus, the loop iteration overheads become much more prominent in Figure 4.3(a).

4.4 TPC-H Margin UDF

In our final experiment, we evaluate the performance of our compilation on a TPC-H-based UDF that was used by Duta et al. [21]. This function takes each part from the TPC-H `PART` table and finds buy and sell dates for the part that produces the greatest profit based on the orders table. The margin benchmark as given in [21] is as follows in Listing 8.1. We ran this experiment on a TPC-H dataset with increasing scale factors from 1 to 10. A TPC-H dataset with a scale factor of 1 is 1GB in size and has several million elements in each table. All these table size metrics increase linearly with the scale factor.

As Apfel does not support cursor loops, the original formulation of this benchmark as in [21] emulates the iteration of a cursor loop by repeated invocations of queries with an order by and a tuple limit of 1. This iteration style comes at the cost of all overheads of a SQL scan for each resulting tuple in the join query in line 15. We reimplemented this function with a more efficient cursor loop and is still supported by our UDF compilation framework and evaluated it alongside. In Figure 4.4, we compare this function's performance (E1), the original margin function's compiled performance (E2), and the original function's inlined (E3) performance on varying TPC-H scale factors.

We see that E3 always performs at least 2x slower than E2 and 15x - 20x slower than E1. We can attribute the performance difference between E2 and E3 again to the fact that E3 involves a CTE table inserted into and scanned from with every iteration of the for a loop. Another important observation is the performance gain from the cursor loop query in E1. Unlike Apfel, the compilation framework's support for E1 shows another advantage of compilation at the execution level over compilation at the SQL level. There are non-trivial existing inlinable SQL constructs that can efficiently emulate a cursor loop's operation beyond making a custom aggregate as in [28]. Compared to the other queries, the speedup on the cursor loop method indicates how the additional native support of cursor loops in our UDF compilation framework allows for more efficient UDFs to be supported. Furthermore, the speedup of E2 with respect to E3 provides another example of the overheads of executing loops in PL/PGSQL via CTEs.

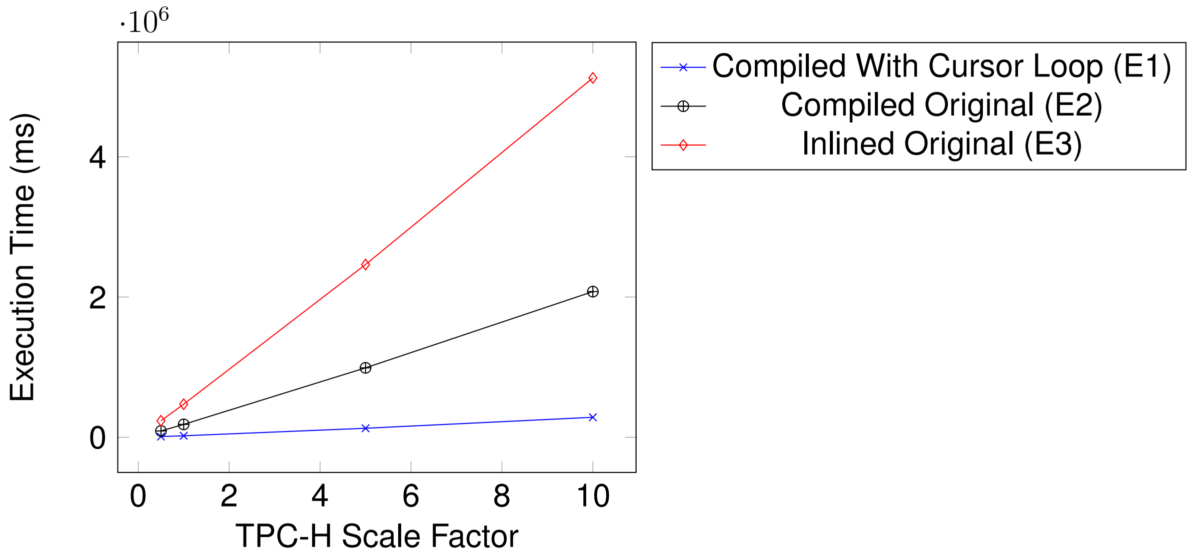


Figure 4.4: Results for Margin Benchmark

4.5 Summary

Our analysis on our experiments suggests that the compiled equivalent of iteration outperforms the inlined equivalent as the former carries much less computational overhead. This difference in overhead is more apparent as the number of iterations in each experiment grows. This trend highlights how the inlined iteration's overhead accumulates with each iteration. Through profiling our experiments, we found that this penalty included all startup costs of a scan and insertion, such as setting up scan buffers. Lowering these overheads by changing configurations does not mitigate the issue as it just lowers the factor by which the inlined and compiled iterations diverge.

Chapter 5

Related Works

There is a large corpus of previous works on inlining UDFs and query compilation. We discuss related works for each of these categories.

5.1 UDF Inlining

Ramachandra et al., developed a technique that compiles scalar T-SQL User Defined Function queries into SQL subexpressions that can be inlined into its caller [42]. Ramachandra et al. detail the process of compiling T-SQL functions into inlinable relational expressions. This is done through a framework called Froid that they implemented in Microsoft SQL Server 2019. This allows the query optimizer to operate across the inlined T-SQL function and its calling query, effectively doing similar optimizations that a standard imperative optimizing compiler would do in most cases. These include dead code elimination, constant propagation, and dynamic slicing. The key idea is that successive imperative regions in T-SQL are chained together using the SQL Server `APPLY` operator. The conversion process is divided into three main phases after parsing the T-SQL function. First, the function is divided into regions that are supersets of basic blocks and can contain conditional regions. There are two different types of regions, (1) sequential and (2) conditional. Conditional regions contain `IF..THEN..ELSE` statements while sequential ones do not. This classification helps with the next phase, where the authors map the individual regions to relational expressions. This is done on a case-by-case basis regarding the type of the imperative T-SQL statement that is being mapped. For example, `IF..THEN..ELSE` statements are mapped to a SQL `CASE` expression. Some imperative statement types, such as loops, do not have as trivial relational statement mappings. These constructs are deemed to be unsupported by Froid. These sets of relational regions are then chained together using the `OUTER APPLY` operator. After this, a client can expect to have one single relational SQL statement to represent the entire query. Now the optimizer can do single statement transformations to originally multiple T-SQL statements, resulting in optimizations equivalent to that of an imperative compiler. Froid does not support recursive functions either as the resulting statement size would blow up endlessly.

Ramachandra et al. evaluated this framework on many customer workloads from the Azure SQL database, and found that Froid supported around 59.8% of the scalar UDFs present in the workload set [41]. They show that Froid yields performance benefits from 5 to 1000x on the supported workloads. This is largely gained because converting imperative T-SQL to relational SQL bypasses the need for context switching to and from the T-SQL imperative interpreter. Inlined queries also benefit from the query optimizer's ability to effectively optimize the body and caller of each inlined T-SQL function.

Duta et al. [21] focus on bridging the semantic disconnect between relational and imperative languages in Postgres by building on the work of Froid's creators. They also look to inline UDFs into relational SQL but through a different method that is amenable to loops. The authors made a framework called Apfel on PostgreSQL that does this transformation from PL/PGSQL to SQL. The main result that Apfel achieves with its inlining method is reducing the cost of context switches that happen upon UDF execution. Such context switches between imperative interpretation and relational query execution are expensive, especially when iterative or recursive UDFs are involved. The main idea to bring iterative and recursive constructs into the world of inlinable queries is with the **`WITH RECURSIVE`** operator. Apfel achieves this by compiling PL/SQL functions through a multitude of intermediary transformations. First, these functions are converted to a Static Single Assignment (SSA) form in the same way that a standard compiler would. Next, this SSA is converted to a functional tail-recursive form called Administrative Normal Form (ANF), which is an intermediary representation for functional compilers. This transformation turned iterative constructs into a set of mutually recursive functions. This set of mutually recursive functions is then converted into one large recursive SQL UDF. In this transformation, nested `Let..Then` ANF constructs that corresponded to successive imperative statements in the original UDF are chained together in the SQL UDF using the **`LATERAL JOIN`** operator. This recursive UDF can now technically be executed by a PostgreSQL DBMS but with the same context switch inefficiencies as before. However, this recursive UDF can easily be converted into an inlinable SQL expression that involves the **`WITH RECURSIVE`** operator. Duta et al. also recognized that the **`WITH RECURSIVE`** operator can be optimized further to emulate loops. They created the **`WITH ITERATIVE`** operator that takes spawns a temporary table, populates it with an initial set of tuples and the iteratively replaces the temp table with the output of a secondary query. This differs from the **`WITH RECURSIVE`** operator in that the **`WITH ITERATIVE`** operator replaces the temp table with its secondary query output instead of adding to the temp table.

Duta et al. proposed this way to convert UDFs with arbitrary control flow structures to inlinable SQL queries. On this workload, this framework showed to have gained runtime savings of approximately 43%. This shows how significant the imperative to relational context switch overheads are in iterative UDFs. This framework does not support special imperative constructs such as exceptions, table value functions, dynamic SQL queries, or ones that modify tables. Duta and Grust went on to extend this line of thought to convert recursive PL/PGSQL functions to recursive CTE's by analyzing their call graphs and add support for recursive UDFs [20].

El-Helw et al. propose an optimization framework on CTEs that involve adapting traditionally imperative optimizations such as common subexpression elimination to this context [22]. Park et al. extend this work to optimizing imperative constructs UDF SQL constructs as well [38]. Park et al.'s technique involves applying known imperative optimization techniques such as loop-invariant code motion on the SQL blocks on these constructs. Floratos et al. take a different approach to optimizing inlinable CTEs by making a framework called SQLoop [24]. This framework ingests recursive or iterative CTE queries passed to an underlying DBMS and processes the query by spawning multiple parallel processes in the underlying DBMS.

5.2 Query and UDF Compilation

The first work on code generation was for IBM System R in the 1970s [16]. There has also been considerable work on the compilation as a mode of query execution in in-memory DBMSs in the past ten years.

Krikellas et al. creates C code templates for each SQL operator to dynamically create source code for each incoming SQL query in a framework called HIQUE [31]. Krikellas et al. point out that iterator-based processing has overheads subsumed by I/O latencies in disk-based systems but are problematic in the world of in-memory database management systems. Interpreted execution has overheads from virtual calls, branch mispredictions, instruction cache misses, and register flushing that is not scalable in this realm. Krikellas et al. proposed that dynamically generated query code will move all costs of interpreting the query plan to the query compilation phase. Neumann further improved upon this notion of code templates by making a more optimized framework that blurs boundaries between operator code templates and makes generated query-code centered around pushing tuples across operators [37]. Neumann proposes compiling generated LLVM IR for each query plan and then executing that. This is done by generating LLVM IR for each query and then compiling that to machine code. Neumann details this process that relies on the concept of dividing the query tree into pipelines. A pipeline is a set of connected components in the query operator tree where a tuple being run through this component can stay in the registers the whole time. For example, a hash join would mark a pipeline boundary on its build side as tuples from the build side have to be aggregated in the hash join table before proceeding upwards. On the other hand, this hash join operator is not a pipeline boundary for the probe side as tuples coming from this side of the query operator tree can be passed upwards and stay in the registers if they find a match join table. This way, one single pipeline can be represented as one function that has a loop. Within a pipeline, an operator will have its parent's generated code nested within its own. Neumann proposes turning the iterator-based model that "pulls" tuples from child operators to one that "pushes" tuples to its parents. As queries are converted to LLVM IR before being converted to machine code, the LLVM framework can optimize this IR through standard and state-of-the-art optimization passes that the LLVM framework provides to optimize the query execution even further. Furthermore, once the compilation is done for a query, the compiled module can be cached for repetitions of this query.

Other systems have adopted this data-centric model, such as NoisePage [39]. Menon et al. further built on this model by allowing "staging points" between operators where batches of tuples are materialized in the generated code to allow for inter-tuple parallelism through SIMD or prefetching [33].

PostgreSQL added LLVM JIT compilation for expressions in Postgres 11 [32]. In a disk-based DBMS such as Postgres, expression interpretation presented an overhead that JIT compilation alleviated. Schüle et al. extended this compilation framework and in [43] propose an extension on the PostgresSQL execution engine that allows it to have a broader scope of supported table value functions. Schüle et al. took advantage of PostgresSQL 11's addition of LLVM JIT compilation of expressions by implementing a type for **LAMBDA** functions. In conjunction with this new lambda type, they also implemented two types to represent multi-rowed subquery expressions called **LAMBDACURSOR** and **LAMBDATABLE**. The former can be thought of as an iterator through the given subquery results and is suitable for single-pass applications. However, the latter materializes all the relevant subquery results and is useful for functions that will incur multiple passes over the data. The paper insists that these three new types, **LAMBDA**, **LAMBDACURSOR**, and **LAMBDATABLE**, synergize to have essential data mining applications in the form of server-side execution, achieving comparable performance to hardcoded data-mining operators. The framework wrote user-defined functions in C stored as a shared library on the database server. One example of such a function would be to label a set of points based on a function on their x and y coordinates. In this case, the function takes in a **LAMBDACURSOR** over the concerned table. It also takes in a **LAMBDA** function that specifies the labeling function and applies it to each row from the cursor. The authors took the LLVM IR of this UDF and injected the LLVM IR of the lambda argument's body passed in at runtime. This allows optimization across the body of the lambda function and the body of the UDF. In this case, Schüle et al. made a tradeoff with compilation time.

UDF compilation has been explored in the context of UDF languages. For example, Essertel et al. discuss speeding up Apache Spark queries by compiling them to native code. They also mention compiling Scala UDFs to the same medium [23]. Crotty et al. discuss an analytical system called Tupleware that discuss LLVM compilation for UDFs written to interface with their custom API [18]. TupleWare analyzes a UDFs LLVM IR to estimate compute and memory load times for the UDF to decide whether map operations in a function should be pipelined or vectorized. Both of these differ from our work. Our work discusses the ramifications of compiling PL/PGSQL in a way that can be generalized to other relational procedural languages such as T-SQL.

Oracle supports native UDF compilation by compiling PLSQL UDFs to C starting from Oracle Database 11g [35]. Similarly, SingleStore DB supports native compilation of a custom procedural UDF language called MPSQL [9]. This system compiles MPSQL functions and other SQL queries into its domain specific language called MPL that is further compiled into machine code. Myers et al. does similar query compilation but into a DSL that is meant to be run in a distributed parallel setting [34].

Chapter 6

Future Work

In this work, we focused on the implementation and performance of UDFs with loops as those constructs have counterparts in the world of UDF inlining. However, we remain curious about compiling other imperative constructs that we identified in Chapter 2 are frequently used but not inlinable as of now. It would be interesting to explore inlining possibilities of these and compare them with their compiled counterparts.

Another interesting direction is to investigate when inlining UDFs would be more optimal than compiling. In this thesis, we did not fully consider the effect of an optimizer on our experiments. Although our experiments compared against optimized executions of inlined queries, it could be the case that this definition of optimized changes with the cost models or table sizes. Inlined UDFs would execute differently with these changes. However, that is not the case for compiled queries. One can go along this line of thought to see how the role of an optimizer affects the comparison between the two UDF execution techniques. Furthermore, the performance of an inlined query is highly dependent on the optimizer implementation [42]. The Postgres optimizer misses out on many contextual optimizations on CTEs [22]. This played a role in our experiments as it affected how optimized were the inlined queries we compared against. Doing similar comparisons with different optimizers could yield interesting results.

On the other hand, future works can investigate optimizer passes in a UDF compilation framework. Although the LLVM framework does low-level optimizations to a compiled UDF in our framework, future works can look into adding higher level optimization passes at the UDF language level. Such passes could include coalescing two SQL assignments together via subquery-inlining as in [38].

Chapter 7

Conclusion

This work discusses a new framework of UDF compilation and evaluates its applicability and performance compared to other UDF query inlining. We found that UDF compilation supports a broader variety of constructs but also outperforms other approaches on our iteration-heavy benchmarks. We find that UDF compilation's more comprehensive language support allows for more efficient functions to be executed. Compilation is bound by the overheads of the individual operators that we are rewriting the UDF too. As a result, UDF compilation rewrites to TPL operators that are much more fine-grained than the SQL operators that UDF inlining frameworks produce. As a result, compilation naturally seems to outperform query-inlining by 2x - 120x depending on the function's nature.

Bibliography

[1] HyPer. https://hyper-db.de.

[2] Information builders, inc. history. http://www.fundinguniverse.com/company-histories/information-builders-inc-history/.

[3] NoisePage. https://noise.page.

[4] A (not so) brief but (very) accurate history of pl/sql. http://oracle-internals.com/blog/2020/04/29/a-not-so-brief-but-very-accurate-history-of-pl-sql/.

[5] SingleStore. https://www.singlestore.com/.

[6] An overview of transact-sql support. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocente

[7] What's new in postgresql 9.3. https://wiki.postgresql.org/wiki/what Technical report, Postgres, 2013.

[8] Pl/pgsql - sql procedural language. https://www.postgresql.org/docs/9.6/plpgsql.html. Technical report, PostgreSql, 2020.

[9] Singlestore documentation. Technical report, SingleStore, 2020.

[10] Transact-sql reference. https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver15. Technical report, Microsoft, 2020.

[11] Understanding sql subqueries, Feb 2020. URL https://www.w3resource.com/sql/subqueries/understanding-sql-subqueries.php.

[12] Oracle: User-defined functions. Web, 2021.

[13] Postgresql: User-defined functions. Web, 2021.

[14] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742797. URL https://doi.org/10.1145/2723372.2742797.

[15] William D Assaf. Subqueries (sql server) - sql server, Feb 2018. URL https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries?view= subquery is a query,SQL Copy.

[16] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu,

Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system r. *Commun. ACM*, 24: 632–646, October 1981.

[17] Microsoft Corp. Sql server: User-defined functions. Web, 2019.

[18] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, abs/1406.6667, 2014. URL `http://arxiv.org/abs/1406.6667`.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[20] Christian Duta and Torsten Grust. Functional-style sql udfs with a capital 'f'. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1273–1287, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389707. URL `https://doi.org/10.1145/3318464.3389707`.

[21] Christian Duta, Denis Hirn, and Torsten Grust. Compiling pl/sql away, 2019.

[22] Amr El-Helw, Venkatesh Raghavan, Mohamed A. Soliman, George Caragea, Zhongxian Gu, and Michalis Petropoulos. Optimization of common table expressions in mpp database systems. *Proc. VLDB Endow.*, 8(12):1704–1715, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824068. URL `https://doi.org/10.14778/2824032.2824068`.

[23] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 799–815, USA, 2018. USENIX Association. ISBN 9781931971478.

[24] S. Floratos, Y. Zhang, Y. Yuan, R. Lee, and X. Zhang. Sqloop: High performance iterative processing in data management. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1039–1051, 2018. doi: 10.1109/ICDCS.2018.00104.

[25] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. *SIGMOD Rec.*, 30(2):571–581, May 2001. ISSN 0163-5808. doi: 10.1145/376284.375748. URL `https://doi.org/10.1145/376284.375748`.

[26] Ahmad Ghazal, Dawit Seid, Alain Crolotte, and Mohammed Al-Kateb. Adaptive optimizations of recursive queries in teradata. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 851–860, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312479. doi: 10.1145/2213836.2213966. URL `https://doi.org/10.1145/2213836.2213966`.

[27] G. Graefe. Volcano— an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994. ISSN 1041-4347. doi: 10.1109/69.273032. URL `https://doi.org/10.1109/69.273032`.

[28] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. Optimizing cursor loops in relational databases. 2020.

[29] A Milton Jenkins and Ron Weber. Using dbms software as an audit tool: The issue of independence. *Journal of Accountancy (pre-1986)*, 141(000004):67, 1976.

[30] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, September 2018. ISSN 2150-8097. doi: 10.14778/3275366.3284966. URL `https://doi.org/10.14778/3275366.3284966`.

[31] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE, 2010.

[32] Dmitry Melnik, Ruben Buchatskiy, Roman Zhuykov, and Eugene Sharygin. Jit-compiling sql queries in postgresql using llvm. In *Proc. PostgreSQL Conf.*, 2017.

[33] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11:1–13, September 2017.

[34] Brandon Myers, Daniel Halperin, Jacob Nelson, Mark Oskin, Luis Ceze, and Bill Howe. Radish: Compiling efficient query plans for distributed shared memory. Technical report, Technical Report, 2014.

[35] Arup Nanda. Oracle database 11g: The top features for dbas and developers. Technical report, Oracle Corp., 2007.

[36] Tanuj Nayak. udf-compilation. `https://github.com/tanujnay112/udf-compilation`, 2020.

[37] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[38] Kisung Park, Hojin Seo, Mostofa Kamal Rasel, Young-Koo Lee, Chanho Jeong, Sung Yeol Lee, Chungmin Lee, and Dong-Hun Lee. Iterative query processing based on unified optimization techniques. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 54–68, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3324960. URL `https://doi.org/10.1145/3299869.3324960`.

[39] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017.

[40] Karthik Ramachandra and Kwanghyun Park. Blackmagic: Automatic inlining of scalar udfs into sql queries with froid. *Proceedings of VLDB*, 12(12):1810–1813, August 2019. URL `https://www.microsoft.com/en-us/research/publication/blackmagic-automatic-inlining-of-`

[41] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. Optimization of imperative programs in a relational database. *CoRR*, abs/1712.00498, 2017. URL `http://arxiv.org/abs/1712.00498`.

[42] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, Cesar Galindo-Legaria, and Conor Cunningham. Optimization of imperative programs in a relational database. 2019.

[43] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. Freedom for the sql-lambda: Just-in-time-compiling user-injected functions in postgresql. In *32nd International Conference on Scientific and Statistical Database Management*, SSDBM 2020, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388146. doi: 10.1145/3400903.3400915. URL `https://doi.org/10.1145/3400903.3400915`.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

Chapter 8

Appendix

```sql
CREATE FUNCTION margin(partkey integer) RETURNS float AS
$$
    DECLARE
    this_order record;
    buy            integer          := NULL;
    sell           integer          := NULL;
    margin         numeric(15,2) := NULL;
    cheapest       numeric(15,2) := NULL;
    cheapest_order integer;
    price          numeric(15,2);
    profit         numeric(15,2);
    tmp_d date;
    BEGIN
    -- first order for the given part
    SELECT o.o_orderkey, o.o_orderdate into this_order
        FROM   lineitem AS l, orders AS o
        WHERE  l.l_orderkey = o.o_orderkey
        AND    l.l_partkey  = partkey
        ORDER BY o.o_orderdate
        LIMIT 1;
    -- hunt for the best margin while there are more orders to consider
    WHILE this_order.o_orderkey IS NOT NULL LOOP
        -- price of part in this order
        SELECT MIN(l.l_extendedprice * (1 - l.l_discount) * (1 +
        ↪  l.l_tax))
                                into price
                FROM   lineitem AS l
                WHERE  l.l_orderkey = this_order.o_orderkey
```

```
                    AND    l.l_partkey  = partkey;
            -- if this the new cheapest price, remember it
            IF cheapest IS NULL THEN
                cheapest := price;
            END IF;
            IF price <= cheapest THEN
                cheapest       := price;
                cheapest_order := this_order.o_orderkey;
            END IF;
            -- compute current obtainable margin
            profit := price - cheapest;
            IF margin IS NULL THEN
                margin := profit;
            END IF;
            IF profit >= margin THEN
                buy    := cheapest_order;
                sell   := this_order.o_orderkey;
                margin := profit;
            END IF;
            tmp_d = this_order.o_orderdate;
            -- find next order (if any) that traded the part
            SELECT o.o_orderkey, o.o_orderdate into this_order
                FROM   lineitem AS l, orders AS o
                WHERE  l.l_orderkey = o.o_orderkey
                AND    l.l_partkey  = partkey
                AND    o.o_orderdate > tmp_d
                ORDER BY o.o_orderdate
                LIMIT 1;
        END LOOP;
        RETURN margin;
        END;
    $$
    LANGUAGE PLPGSQL;
```

Listing 8.1: Margin Function. This finds the the best profit margin one can obtain by buying a an order of a given part from the TPC-H `ORDERS` table and selling on a later date and returns that profit margin.