Mathematish-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Computer Science

From PL/pgSQL to C: What Are the Optimizations?

Alexander Phi. Goetz

March 31st 2023

Examiner

Prof. Torsten Grust

Supervisor

Denis Hirn

Alexander Phi. Goetz:
From PL/pgSQL to C: What Are the Optimizations?
Bachelorthesis Computer Science
Eberhard Karls Universität
From November 1st 2022 to March 31st 2023

Selbständigkeitserklärung

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

Ort, Datum	Alexander Phi. Goetz

Acknowledgement

First of all thank you Denis! As my supervisor you were always reachable when I ran head first into a wall and had no idea how to continue.

Also a ’thank you’ to the whole DB Research Group! I do not want to know how many hours I sat in your offices and worked on this thesis.

Abstract

PL/pgSQL is an imperative language extension that is implemented as an interpreter layered on top of PostgreSQL. This introduces friction during execution because, for each embedded statement, an executor must be instantiated, run, and finally purged.

In this thesis, we investigate what happens when we translate PL/pgSQL functions into equivalent `C` functions, thus eliminating the interpretation overhead of the PL/pgSQL interpreter.

Contents

Acronyms

DDL	Data Definition Language
DML	Data Manipulation Language
OID	Object Identifier
PL/pgSQL	PL/SQL PostgreSQL variant
PL/SQL	Procedural Language extension to the Structured Query Language
RDBMS	Relational Database Management System
SPI	Server Programming Interface
SQL	Structured Query Language
TPC-h	Transaction Processing Performance Council - Decision Support
UDF	User Defined Function

1 Introduction

PostgreSQL allows for an procedural dialect called PL/SQL to be used in User Defined Functions. This dialect has several imperative control flow constructs, like loops, conditionals and statement-based evaluation, supported within its language. An important feature however is that the user can use normal SQL queries as value-returning expressions. Where a procedural or imperative-versed developer might benefit from this kind of expressiveness the resulting functions then live in the worlds of PL/SQL and standard SQL query execution. Switching between those two contexts takes time.

Following the idiom of moving computation close to the data, the works [1][2][3] have shown that by compiling PL/SQL into standard SQL queries. By employing recursive Common Table Expressions and everything SQL:1999 has to offer they were able to remove any need to switch contexts at all. Having a sole SQL query yielded a significant runtime reduction.

Where those works removed the imperative style altogether this work implements the same PL/SQL UDFs within a C language extension for PostgreSQL. Building an extension directly onto PostgreSQL's own data types and functions allows us to use the same functions and procedures the PL/SQL version would. As well as shortening trivial operations by implementing them directly in C.

This might suggest a similar approach of removing the need for context switches but this work aims to identify possible overhead imposed by the PL/pgSQL interpretation of PostgreSQL. The context switches remain as the embedded queries were not translated to C but still rely on the power of the PostgreSQL executor.

Having no parsing and interpretation overhead should improve the runtime of the C version compared to the original. What we found is that this advantage is minimal. We measured that the C language implementations are around one to five percent faster than the original PL/pgSQL versions. In more complex use cases of academic interest, like Marching Squares, the original version outperforms our approach.

2 Basics

Before jumping straight into the feat of translating a UDF from PL/pgSQL into C we should understand some basic acronyms and terms.

2.1 SQL

The SQL is *the* established language to communicate with most every Relational Database Management System (RDBMS). As it is used to manage data and structure of databases it can be subsected into more domain-specific aspects:

- The Data Definition Language (DDL) allows the creation, alteration and dropping of tables, functions, data types, indices and so on.
- Whereas the Data Manipulation Language (DML) fills the structure defined with the DDL with information. Also being able to modify and select said information from the database.

Whilst SQL itself is standardized and has six revisions with `SQL:2006` being the latest, every RDBMS implementation differs the farther the functionality strays from the basic `SELECT-FROM-WHERE` clauses.

With every RDBMS having a different flavor and level between standards. The system and version used in this work is PostgreSQL14.0.

2.2 PostgreSQL

This work is only possible through the Database it is explored on. PostgreSQL is the RDBMS implementing the functionality expressed in the SQL standard. It is based on the works of the POSTGRES Project at the Berkeley Computer Science Department of the University of California. PostgreSQL ows its nickname Postgres this hereditary basis.

The current character of PostgreSQL as an Open Source Project allows for a wide range of distributions on several platforms. Coinciding with this large user base the project provides solid end-user documentation [4]. Having an OSP makes it possible to take a deep dive into its inner workings. This is especially fruitful for us as this work attempts to formulate functions that integrate structures found in the implementation of the engine.

2.3 PL/SQL

PL/SQL or in this work PL/pgSQL – the *pg* standing for *PostgreSQL* is a SQL language extension that allows users to interact with the database procedurally and imperatively. PostgreSQLsupports its variant with different imperative language constructs. Notable features are *loops*, *Conditionals* and *Variable Assignment*.

Normal SQL queries can be used inside each of the expressions. For example one could assign query q to a variable v like `v := (Q);`. Furthermore. Every SQL expression e is a valid expression in PL/pgSQL. [5]

2.4 User Defined Functions

User Defined Functions are exactly what the name suggests. A user of the RDBMS is able to define a function that consumes several arguments.

The naive attempt would be to take a Query q that either constant or requires an argument (see Listing 2.1) and put it into a UDF f and from then on instead of having to write out q the user can simply call it with `SELECT f(100);`.

```
1  CREATE OR REPLACE FUNCTION f(n INTEGER)
2  RETURNS INTEGER AS
3  $$
4      SELECT SUM(x)
5      FROM   generate_series(1, n) AS x
6  $$
7  LANGUAGE SQL STRICT;
```

Listing 2.1: A plain SQL UDF

They can also be used to return a table result such as the built-in function `generate_series` as used in Listing 2.1 Line 5. Such UDFs allow for the compartmentalization of complex queries and the reusability of reoccurring code snippets across the database.

Other than plain PostgreSQL style SQL PostgreSQL supports other language styles to be used in an UDF definition. Such languages are Python, Perl and most notably the aforementioned PL/pgSQL since Release 6.4 [6]:

```
CREATE OR REPLACE FUNCTION f'(n INTEGER)
RETURNS INTEGER AS
$$
DECLARE
    sum INTEGER := 0;
BEGIN
    FOR i IN 1..n LOOP
        sum := sum + i;
    END LOOP;

    RETURN sum;
END
$$
```

```
LANGUAGE PLPGSQL STRICT;
```

Listing 2.2: A simple PL/pgSQL UDF

2.5 Server Programming Interface

Now that we have established the use of User Defined Functions it should not surprise that there are further ways to adapt the DBMS to fit a user's requirements. One of those is to develop and use PostgreSQL extensions that plugin right at the server engine level. The preferred language implementing these in is C. For the developers convenience PostgreSQL ships with *Makefiles* for the task.

To query knowledge from tables inside the database the developer has to use the Server Programming Interface (SPI).

To get knowledge from the database the respective function first has to initiate the connection to the database using the `SPI_connect` command. The next step is to formulate the query that results in the required information. Similar to normal database interaction these queries are formulated in SQL. Of cause, they can depend upon input arguments. Therefore we prepare statements. Using SPI we prepare a cursor that results in a plan that can be executed later on.

Once the holes in the plan can be filled with known values these are used to execute the query. At this point, it is imported to know any arguments that are `NULL`. The results are then found in the `SPI_tuptable` To get the values they are accessed with `SPI_getbinval` and by then it is possible to find out what the desired value is or if the value is null.

2.6 What are Object Identifiers (OIDs)?

In PostgreSQL, everything has an Object Identifier (OID). From tables to data types, from functions to operators and roles. Everything is managed through those identifiers. This is important concerning using any of the types, functions and operators et cetera a user may have provided themselves.

One could find out the OID manually by looking it up in the fitting system catalog `pg_`*catalog* [7]. There are catalogs for everything. From *A* like `aggregate` to *ƵU* for `user_mapping`. For the extent of this work, we were only interested in the catalogs for *Types*, *Operators* and *Procedures*.

This rather manual labor takes time and "Only works on my machine" as the database-wide identifiers are assigned after `CREATE`-statements of the respective definitions. A more suited way to get hold of the identifiers is to use the internal `to_reg`*catalog*-function that can be used to lookup the function with its given name. As this function is not a directly callable C function but rather a `PG_FUNCTION` as is the one being implemented one must use the provided interface. Using a `DirectFunctionCall1` with the aforementioned built-in and the name of the desired object as arguments one can recover the used OID at the time of `CREATE EXTENSION`.

3

From PL/pgSQL to C

In the previous Chapter we introduced the concept of procedural language UDFs. This Chapter is dedicated to exploring the several aspects that will have to be taken into consideration when and how a translation of the PL/pgSQL dialect into the C language is achieved.

Where we previously looked at a toy example of a PL/pgSQL UDF the example in Listing 3.1 provides a more real world example. The function identifies suppliers that did not successful fulfill orders in a timely manner.

In this function we identify multiple aspects that invoke a sense of C-like imperative style motivating the task of translating it into pure C.

```
DROP FUNCTION order_kept_waiting(int,int);
CREATE FUNCTION order_kept_waiting(suppkey int, orderkey int)
RETURNS boolean AS
$$
  DECLARE
    lis   lineitem[];
    li    lineitem;
    blame boolean := false; -- is suppkey to blame?
    multi boolean := false; -- does this order have multiple suppliers?
  BEGIN
    lis := (SELECT array_agg(l)
            FROM   lineitem AS l
            WHERE  l.l_orderkey = orderkey);
    FOREACH li IN ARRAY lis LOOP
      multi := multi OR li.l_suppkey <> suppkey;
      IF li.l_receiptdate > li.l_commitdate THEN
          IF li.l_suppkey <> suppkey THEN
            RETURN false;
          ELSE
            blame := true;
          END IF;
      END IF;
    END LOOP;
    RETURN multi AND blame;
  END;
$$
LANGUAGE PLPGSQL;
```

Listing 3.1: PL/SQL reimplementation of TPC-H query Q21

```
τ f(τ₁ ν₁, …, τₙ νₙ)
{
  // [...]
}
```

Listing (3.3) Naïve C function

```
PG_FUNCTION_INFO_V1(f);
Datum f(PG_FUNCTION_ARGS)
{
  Datum ν₁ = PG_GETARG_DATUM(0);
  // [...]
  Datum νₙ = PG_GETARG_DATUM(n−1);
}
```

Listing (3.4) *Magic Function* definition

Figure 3.1: ”Normal” C function definition vs. C function with ”Version 1” calling convention

3.1 Defining Dynamically loaded Functions

When developing C functions that should be callable from other Postgres functions these functions should adhere to the ”Version 1” calling conventions [8]. Meaning they should be exposed like so:

```
CREATE FUNCTION f(ν₁ τ₁, …, νₙ τₙ)
RETURNS τ
AS '$libdir/extension', 'f'
LANGUAGE C;
```

Listing 3.2: SQL DDL for C language functions

`'$libdir/`*extension*`'` is the path to the object file of the extension. Postgres will substitute ’`$libdir'` with the path to the extension folder where the object file to the *extension* resides. f is the name of the function that we want to expose to the user.

The aforementioned object file has to be present before loading the function. Postgres won’t start a compilation process. This has to be manually done.

Whilst loading the function the database checks whether it was compiled for a compatible database. An obvious incompatibility would be different major release versions of Postgres. This requires a magic block to be present in the code. This magic block earns the nickname for these types of functions that are also called *Magic Functions*.

```
PG_MODULE_MAGIC;
```

To access this macro and every other macro or builtin function it is required to include the header files `postgres.h` and `fmgr.h` as well as every other necessary header file that comes up in interaction with builtins.

According to the v1 calling conventions the C function header is not an analogue to the header of Listing 3.2 as the left Listing in Table 3.1 but rather has to uphold the convention with announcing itself with the macro `PG_FUNCTION_INFO_V1` and returning a value of type `Datum`. The n arguments also are replaced by the macro `PG_FUNCTION_ARGS`. To access these Postgres provides the macros `PG_GETARG_`τ where τ is the type of a builtin. These will be discussed later on in Subsection 3.6.0.1.

	`PG_FUNCTION_INFO_V1(`f`);`
	`Datum` f`(PG_FUNCTION_ARGS)`
τ f`(`τ_1 v_1`, …, `τ_n v_n`)`	`{`
`{`	`Datum` v_1 `= PG_GETARG_DATUM(0);`
`// [...]`	`// [...]`
`}`	`Datum` v_n `= PG_GETARG_DATUM(`$n-1$`);`
	`}`

Table 3.1: "Normal" C function Definition vs. *Magic Function* Definition

The right Listing in Table 3.1 only retrieves the arguments as of type `Datum`. Why this is a sensible approach will be discussed in the successive section.

3.2 Handling Datums

When taking in the lines above one might ask themselves *"What is* **`Datum`** *?"* The short answer is *Everything is a* **`Datum`** . And when interacting with builtins it is wise to seldom force them to an actual value. Simply spoken **`Datum`** is the most general representation of data available.

Where in object-oriented languages like Java there is a `Object` class that everything inherits trades from, here we have an `uintptr_t`. It is an *unsigned integer* type that is exactly the size of a pointer. The "size of a pointer" part is important as a **`Datum`** will both be a pointer to more complex types as well as holding values in of itself that may be cast to something like an actual integer or floating value.

Regarding the built-in types, there are functions in the style of `DatumGet`τ and τ`GetDatum` to unpack and pack values. These are the most useful when handling primitive types or for accessing structs. A more thorough dive into Types can be found in Section 3.6.

3.3 Outlining the Control Flow

As PL/pgSQL presents a programming style that allows for complex control flow. So it is no surprise that the translation in C has to find equivalences to those expressions. Luckily both PL/pgSQL and C are based on imperative paradigms.

3.3.1 Declaration, Initialisation and Assignment

One crucial part of imperative style is the extraction of subexpressions into constants and variables. This concept can be split up into three components. First the *declaration* of the variable with its name and type of it. Then the *initialisation* with a value. Normally done by *assigning* it to the variable name. The process of assigning values to the variable can be influenced by other control flow decisions.

PL/pgSQL insist on declaration of variables for each **`BEGIN`**`...`**`END`**`;`-block unless there are no varibles present. Such a block presents itself as shown in Table 3.2.

PL/pgSQL statement	C statement
`DECLARE` *declarations* $name_1$ τ_1; $name_2$ τ_2 := v_2 `BEGIN` *statements* `END;`	`// --- DECLARE --------------` *declarations* `Datum` $name_1$; `//` τ_1 `Datum` $name_2$ = v_2; `//` τ_2 `// ---------------------------` *statements*
v := *expression*;	v = *expression*;

Table 3.2: Translation of Declarations and Assignment

Assignment can happen in both the declaration of the variable and the BEGIN...END;-block. The differences in syntax are marginal as the PL/pgSQL assignment operator (:=) gets virtually replaced by the standard C assignment operator (=).

3.3.2 Condtitionals

To be able to construct more complex control flow branching is necessary. PL/pgSQL allows its users the use of IF to enable decisions dependend on boolExps. Those expressions can be combined using Logical Conjunctions (AND) and Disjunctions (OR).

Table 3.3 shows the direct translation of the conditionals. Where C encapsulates the *boolExp* with parenthesis and the *statements* by opening a new scope using brackets, the PL/pgSQL captures the expression between the IF and THEN. The statements are themselves scoped by THEN and END IF;

Branching with ELSE is done simply by placing the keyword between THEN and END IF;. Similar to the C translation.

The option to require another boolean expression to be true for the ELSE branch is possible using ELSIF instead and putting the new boolean expression between it and THEN. In C this keyword is an extension of **else** by starting a new **if** in a bracketless scope.

3.3.3 Loops

Coming to loops. PL/pgSQL supports several flavors of them [9]. In this work, we encountered the loop constructs shown in Table 3.4.

Beginning with the option for infinite loops using solely the LOOP construct. To exit these the control commands BREAK and CONTINUE are also present and found their direct counterparts. Whilst infinite loops are interesting most use cases require *finite* loops. Starting with WHILE loops that – like most programming languages – require a boolean expression that evaluates to TRUE to run.

Next, we have the FOR loop. Allowing for more fine control over looping over integer types. Indifferent from the textbook definitions of **for** loops the values of $start, name, end \in \mathbb{Z}$ with $start \leq name \leq end$ or in the case of REVERSE where $start > end$, $start \geq name \geq end$. The

PL/SQL	C
boolExp `AND` *boolExp*	*boolExp* `&&` *boolExp*
boolExp `OR` *boolExp*	*boolExp* `||` *boolExp*
`IF` *boolExp* `THEN` *statements* `END IF;`	`if` (*boolExp*) { *statements*
`IF` *boolExp* `THEN` *statements* `ELSE` *statements* `END IF;`	`if` (*boolExp*) { *statements* } `else` { *statements* }
`IF` *boolExp*$_1$ `THEN` *statements* `ELSIF` *boolExp*$_2$ `THEN` *statements* `END IF;`	`if` (*boolExp*$_1$) { *statements* } `else if` (*boolExp*$_2$) { *statements* }

Table 3.3: Translation of Conditionals (Conjunction, Disjunction & Branching)

stepsize also is controllable using the argument *step*. The comparison between the languages is observable in Table 3.4.

The last kind of loop encountered in this work is the `FOREACH` loop which iterates over SQL arrays. The keyword `ARRAY` annotates this as well. Table 3.4 presents a lot of boilerplate compared to the previous translations. Where the PL/pgSQL variant is compact and concise in its usage. Meaning the user solely has to provide the array and the new v its elements should be bound to; the C translation requires a lot more knowledge about the array. `typlen`, `typbyval` and `typalign` will hold information about the data type stored in the array. `get_typlenbyvalalign` retrieves these datapoints. For further information about Arrays refer to Section 3.8. While this allows knowledge of the contents structure, iteration or looping over it is of the domain of the `array_iter` struct. To iterate over it we employ a `for` loop that increments a run variable that is then used in `array_iter_next`. This returns the desired v element.

3.4 Querying the Database

PL/pgSQL supports embedded SQL queries as both variable assigned and embedded expressions. For the embedded expressions we have to extract a descriptive name and convert it to a Meta-Variable. From there every query is of the schema $name$ `:=` $(q)[v_1, \ldots, v_n]$ where v_i represent free variables present in query q. Their values enter from other variables.

We see this behavior in the Original Query in Table 3.5. There is an example such application in PL/pgSQL with the minor difference that in any real-world occurrence will have the `$1` already replaced with the label `orderkey`.

A modification we make to handle query results consistently is to rewrite the projections `SELECT t.*` into ones where instead of multiple columns we return one like `SELECT t`. With this, we can decompose the Row Type (see Section 3.6 later on.

PL/pgSQL loop construct	C translation
`LOOP`	`while (true) {`
`  statements`	`  statements`
`END LOOP;`	`}`
`IF boolExp THEN`	
`  EXIT;`	`if (boolExp)`
`END IF;`	`  break;`
`EXIT WHEN boolExp;`	
`IF boolExp THEN`	
`  CONTINUE;`	`if (boolExp)`
`END IF;`	`  continue;`
`CONTINUE WHEN boolExp;`	
`WHILE boolExp LOOP`	`while (boolExp) {`
`  statements`	`  statements`
`END LOOP;`	`}`
`FOR v IN start..end LOOP`	`for (int v = start; v <= end; v++)`
`  statements`	`{  statements }`
`END LOOP;`	
`FOR v IN start..end BY step LOOP`	
`  statements`	`for (int v = start; v <= end; v += step)`
`END LOOP;`	`{  statements }`
`FOR v IN REVERSE start..end LOOP`	
`  statements`	`for (int v = start; v >= end; v--)`
`END LOOP;`	`{  statements }`
	`array_iter it;`
	`int16 typlen;`
	`bool  isNull, typbyval;`
	`char  typalign;`
	`int nitems = DatumGetInt32(`
	`  DirectFunctionCall1(`
	`    array_cardinality, v_arr));`
`FOREACH v IN ARRAY v_arr LOOP`	`get_typlenbyvalalign(`
`  statements`	`  ARR_ELEMTYPE(v_arr),`
`END LOOP;`	`  &typlen, &typbyval, &typalign);`
	`array_iter_setup(&it, v_arr);`
	`Datum v;`
	`for (int i = 0; i < nitems; i++ ){`
	`  v = array_iter_next(`
	`    &it, &isNull, i,`
	`    typlen, typbyval, typalign);`
	`  statements`
	`}`

Table 3.4: PL/pgSQL to C better caption

Starting there we can remember the introduction of the SPI in the previous chapter. SPI is needed to connect to the database and send off queries. Therefore the function f has first to connect to it and from then on we can use SPI procedures in the C definition of f.

Every invocation of a query is a two-part process. On the one hand the query plan has to be constructed with possible holes for free variables and on another hand the execution of said plan taking in arguments.

For the setup, we require a `SPIPlanPtr` that holds a prepared cursor, ie. the query q with additional information. When observing Table 3.5's ① Plan Setup this pointer is declared outside of the function. We do this together with `SPI_keepplan` to hold on to the plan after a function call. As the query with holes is a static piece of information there is no need to reconstruct it with every invocation of f. Especially since f might be called a large number of times within a query itself.

Another part of this setup is an array of the length of n where n is the number of free variables in q. The `argtypes` array holds the OIDs of the types of said free variables. How these OIDs can be found out will be discussed in Section 3.6. The query q, the number of free variables n and their types τ_i are all that is needed to construct the plan using `SPI_prepare_cursor`.

For later execution, we also prepare two more arrays in this step. The `values` and `NULLs` arrays. Each is supposed to keep knowledge about the arguments. The first will be filled with the values of type `Datum` for the v_i and the latter sets the markers whether these v_i are of value SQL **`NULL`**.

The execution of the plan requires these arrays to be filled. This happens right away in ② Execution of Plan in Table 3.5. We move the initialization or value assignment close to plan execution both can be dependent on value changes in loops. But with the plan, the value and null store we have everything to invoke `SPI_execute_plan`. The two additional arguments are a read-only flag and the number of rows that should be returned.

`res` holds the response code for the query. The query result will be put into `SPI_tuptable`. To access values another SPI function is used. `SPI_getbinval` accesses the tuptable's values by

3.5 Nullhandling

With the ability to receive values either as arguments or by queries, we are now at a stage where we should address the value **`NULL`** and it's status compared to `NULL` in C.

SQL allows for very easy use of this value. It behaves similarly to `Nothing` of the Maybe Monad from Functional Programming Languages. Everything that comes into contact with it becomes **`NULL`** itself. Meaning that computations can produce a result that may not be of an actual value but one that does not break further computations on it.

In C there is no such luxury. Every error-prone operation that can result in a faulty value has to be handled such that the behavior of the PL/pgSQL **`NULL`** is preserved. That said, having to handle `NULL` in C should be of no surprise as most imperative languages have the concept of it and handling it is vital in these as well.

We saw in the previous section that plan execution requires an array of Null markers and that

Original Query

```
-- With $v_i$ being free variables
name := ($q$)[$v_1, \ldots, v_n$];

-- ie. in order_kept_waiting where orderkey is free
lis := (SELECT array_agg(l)
        FROM   lineitem AS l
        WHERE  l.l_orderkey = $1)[orderkey];
```

① Plan Setup

```
SPIPlanPtr plan_name = NULL;

// ...
Datum f(PG_FUNCTION_ARGS)
{
  // ...
  int ret;

  if ((ret = SPI_connect()) < 0)
    elog(ERROR, "SPI_connect returned %d", ret);

  // plan name
  Oid *argtypes_plan_name = palloc(n * sizeof(Oid));
  argtypes_plan_name[0] = oid($\tau_1$); // Oid of type $\tau_1$
  …
  argtypes_plan_name[n-1] = oid($\tau_n$);

  if (plan_name == NULL)
    plan_name = SPI_prepare_cursor(
      "q", n, argtypes_plan_name, 0);
  SPI_keepplan(plan_name);

  Datum *values_name = palloc(n * sizeof(Datum));
  char  *NULLs_name = palloc(n * sizeof(char));
  // ...
}
```

② Execution of Plan

```
int res;
bool isNull;
Datum name;

for (int i = 1; i <= n; i++)
{
  values_name[i-1] = $v_i$;
  NULLs_name[i-1] = ($v_i$ == NULL) ? 'n' : ' ';
}

res = SPI_execute_plan(plan_name, values_name, NULLs_name, true, 1);
if (SPI_processed <= 0) elog(INFO, "No rows processed");

name = SPI_getbinval(SPI_tuptable->vals[0],
                     SPI_tuptable->tupdesc, 1, &isNull);
if (isNull) name = NULL;
```

Table 3.5: Steps to translate an embedded query q with n free variables v_i

other SPI functions such as SPI_getbinval receive a boolean pointer argument named isNull. Within these functions the value of this flag is set depending if the resulting value that should be NULL.

3.6 Data Types

If we take a look at the example given at the beginning of this chapter with Listing 3.1 we see that there are two variables of type lineitem. In this chapter, we want to explore the creation, composition, decomposition and access of built-in as well as user-defined types.

3.6.0.1 Built-In Data Types

When interacting with the database both in SQL and C we encounter several data types. In SQL these range from the basic integer overspecialized monetary types [10] to geometric types [11]. Another quite common class of types is Date/Time Types [12]. In C we have several number types like integers and floating point values in different bitlengths as well as truth values with bool and char where it is an integer value of one byte in length.

A list of Built-in SQL types with their corresponding C type is found in Table 3.6. Column ”Defined In” shows in which header files each of the respective definitions takes place. To the readers' surprise, the Table solely lists char and possibly bool as a C language built-in. While float4, float8, int16, int32 and int64 are defined in the postgres.h header file they all are aliases for the known C types float, double, short, int and long.

These seven primitive types are also the only ones we consider for in-C-use. Meaning that those are the only types that are an exception to the implicit rule to *”Always work with type Datum”*. In these instances, we retrieve the value for cases that inform control flow and are required for decisions. For the condition in an if statement, we extract the boolean value by calling DatumGetBool(ν). A similar case is made for run variables of loops. How other builtin types are used is explained in the later Section 3.7 Function Calls.

One point previously skimmed over was the oid(τ) placeholder function when preparing the query plan in Section 3.4. The question is where can the Object Identifier be found or retrieved from? The answer to this is the System Catalog pg_type [7]. Every piece of information regarding each type present in the database can be found here.

There are several paths to access the OID through them. When developing it is always an option to interact with the database. In Listing 3.5 are two example queries that result in the OID of a type τ. The first example selects and then projects the OID from the catalog table where the second example calls a builtin function that is then cast to oid.

SQL Type	C Type	Defined In
boolean	bool	postgres.h (maybe compiler built-in)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(compiler built-in)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
float4 (real)	float4	postgres.h
float8 (double precision)	float8	postgres.h
int2 (smallint)	int16	postgres.h
int4 (integer)	int32	postgres.h
int8 (bigint)	int64	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
numeric	Numeric	utils/numeric.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	RegProcedure	postgres.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
timestamp with time zone	TimestampTz	datatype/timestamp.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Table 3.6: Equivalent C Types for Built-in SQL Types (taken from CITE)

```
-- From pg_type catalog table
SELECT t.oid
FROM   pg_type AS t
WHERE  t.typname = 'τ';

-- Function Call with casting regtype to oid
SELECT to_regtype('τ') :: oid;
```

Listing 3.5: Queries to retrieve OID of type τ from the Database

While hardcoding these identifiers may be fine when being constrained to types that the RDBMS is shipped with, the identifiers of User Defined Types are not guaranteed to have the same OID in every database instance or even after their modification. Hence there is a need to dynamically get these values. This is done by utilizing the function used in the second previous example and issueing a Direct Function Call. Those will be thoroughly discussed in Section 3.7.

```
Oid get_typeoid(char *typname)
{
  return DatumGetObjectId(
    DirectFunctionCall1(to_regtype, CStringGetDatum(typname)));
}
```

Listing 3.6: Implementation of function oid

Other pieces of information present in the catalog table are columns designated `typlen`, `typbyval` and `typalign`. They all give insight in the memory representation of the type. Type by value (`typbyval`) is a flag denoting whether the representing `Datum` is used as a pointer or indeed stores the value in itself. For the latter the Type length (`typlen`) is of interest as it stores the bitlength of a given type. Lastly, types are padded to allow for ease of alignment on disc systems. Which of four possible alignments is saved in `typalign`.

For some applications – mainly Arrays (see Section 3.8 for more) – this information is crucial. To access these fields we again have to know the OID. From there we look up the `HeapTuple` in the System Cache that then is converted into a struct that holds the meta information we are after. For convenient use we defined a struct holding these three values and a helper function.

3.6.0.2 Composite Data Types

Coming back to User Defined Types. There are multiple ways how a new type enters the database. The most natural for the user should be table-/row types. By writing the `CREATE TABLE` DDL statement a new type has been created. An example would be the one used to create the `lineitem` table.

```
CREATE TABLE LINEITEM ( L_ORDERKEY    INTEGER NOT NULL,
                        L_PARTKEY     INTEGER NOT NULL,
                        L_SUPPKEY     INTEGER NOT NULL,
                        L_LINENUMBER  INTEGER NOT NULL,
                        -- [ omitted 11 lines ]
                        L_COMMENT     VARCHAR(44) NOT NULL);
```

Listing 3.8: Definition of the `lineitem` type as a table type definition

```c
typedef struct {
    int  elmlen;
    bool elmbyval;
    char elmalign;
} typeinfo;

typeinfo get_typeinfo(Oid elmtype)
{
  HeapTuple tp;
  Form_pg_type typtup;

  typeinfo ti;

  tp = SearchSysCache1(TYPEOID, ObjectIdGetDatum(elmtype));
  if (!HeapTupleIsValid(tp))
    elog(ERROR, "cache lookup failed for type %u", elmtype);
  typtup = (Form_pg_type) GETSTRUCT(tp);
  ti.elmlen   = typtup->typlen;    // int
  ti.elmbyval = typtup->typbyval;  // bool
  ti.elmalign = typtup->typalign;  // char
  ReleaseSysCache(tp);

  return ti;
}
```

Listing 3.7: Retrieval of `typlen`, `typbyval` and `typalign` using the OID

Projecting a row from it would result in a row type of `lineitem` for example:

`(1,1552,93,1,[...],"egular courts above the")`

We find this exact type in the introductory example Listing 3.1.

Another approach to creating a Composite type would be by writing a DDL statement like the one on the left side of Table 3.7. The right side then represents possible implementations within the translation. The upper example may be of more use when the τ_i fall in the previously mentioned seven primitive types, otherwise the motivation to keep values of type `Datum` prevails.

Having a struct suggests that composite values can directly transform themselves into these. This is not the case. In Listing 3.9 it is shown that a `ExpandedRecordHeader` is needed to access the values that then can be either put in use directly or stored within the struct posing as an analog to the SQL variant.

```
$\tau_{new}$ $name_{new}$;

ExpandedRecordHeader *erh = DatumGetExpandedRecord($v_{comp}$);
deconstruct_expanded_record(erh);

$name$.$name_1$ = erh->dvalues[0];
...
$name$.$name_n$ = erh->dvalues[$n-1$];
```

Listing 3.9: Deconstruction of Composite Types

SQL	C
	`typedef struct {`
	τ_1 $name_1$;
	...
`CREATE TYPE` τ_{new} `AS (`	τ_n $name_n$;
$name_1$ τ_1,	`}` τ_{new};
...	
$name_n$ τ_n	`typedef struct {`
`);`	`Datum` $name_1$, // τ_1
	...
	$name_n$; // τ_n
	`}` τ_{new};

Table 3.7: Translation of SQL Composite Types into C Structs

Working with Composite Types not only consists of deconstructing them but also of their construction namely for UDFs that return them. We achieve this by retrieving the result type of the function itself by invoking `get_call_result_type`. This call retrieves the OID and Tuple Descriptor `TupleDesc`. In Table 3.8 it is shown that these bits of information are enough to construct a new Tuple with the desired Composite Return Type τ_{comp}. The values v_i are put into a `Datum` Array. Also there it is required to provide an Array with flags whether the corresponding v_i is `NULL`. Then one can form the `HeapTuple`.

3.7 Function Calls

In the previous Section, we introduced different types. But as said there C functions and operations only work on seven basic types that are native to C themselves. In this Section, we discuss how functions and operations can be used on more complex data types and structures.

The types already provided by PostgreSQLall come with consumer functions that allow for interaction with said types as well as producing functions. The most foundational of these two are τ`_out` and τ`_in`. The latter parses a string to create a value v of type τ, where τ`_out` converts the value v to a string representation.

What other functions are available for any type τ can be looked up in the system catalogs. The system catalog `pg_proc` to be exact. Listing 3.10 shows a query that returns every type τ consuming function. When multiple argument types are known the array in the `WHERE` clause gets expanded.

```
SELECT p.oid, p.proname, p.prorettype :: oid :: regtype,
       (SELECT array_agg(name)
        FROM   unnest(p.proargtypes :: oid[] :: regtype[]) AS _(name)
       ) AS proargtypes
FROM   pg_proc AS p
WHERE  ARRAY[to_regtype('τ') :: oid] <@ p.proargtypes :: oid[];
```

Listing 3.10: Query to list all type τ consumer functions

Similarly, it is possible to list all type τ returning functions by capturing the `prorettype`.

PL/SQL	C

RETURN (v_1, …, v_n) :: τ_{comp};	Datum *values_τ_{comp}
	  = palloc(n * sizeof(Datum));
	bool *NULLs_τ_{comp}
	  = palloc(n * sizeof(Datum));
	
	// Fill Arrays with values/NULL flags
	// for $\{v_i \mid 0 < i \leq n\}$
	values_v_{comp}[$i-1$] = v_i;
	NULLs_v_{comp}[$i-1$] = v_i == NULL;
	
	Oid oid_τ_{comp};
	TupleDesc desc;
	
	get_call_result_type(
	  fcinfo, &oid_τ_{comp}, &desc);
	desc = BlessTupleDesc(desc);
	
	Datum v_{comp} = HeapTupleGetDatum(
	  heap_form_tuple(
	    desc, values_v_{comp}, NULLs_v_{comp}));
	
	PG_RETURN_DATUM(v_{comp});

Table 3.8: Translation of Construction of Composite Types

```
SELECT p.oid, p.proname, p.prorettype :: oid :: regtype,
       (SELECT array_agg(name)
        FROM   unnest(p.proargtypes :: oid[] :: regtype[]) AS _(name)
       ) AS proargtypes
FROM   pg_proc AS p
WHERE  p.prorettype = to_regtype('τ') :: oid;
```

Listing 3.11: Query to list all type τ returning functions

Similar to functions a developer might have additional knowledge about the function they are looking for. Maybe they are not after a function in the first place but rather an operator utilizing this function. And as this operator has a name the System Catalog pg_operator becomes of interest. Especially the column oprcode where the function name is found. querying the catalog where oprname $\equiv$ op can return a lot of entries because of operator overloading.

```
-- Fine-grained control
SELECT opr.oid, opr.oprname, opr.oprcode, ppr.oprresult,
       opr.oprleft  :: oid :: regtype,
       opr.oprright :: oid :: regtype
FROM   pg_operator AS opr
WHERE  opr.oprname  = 'op'
   AND opr.oprresult = to_regtype('τ_resutl') :: oid
   AND opr.oprleft   = to_regtype('τ_l') :: oid
   AND opr.oprright  = to_regtype('τ_r') :: oid;

-- Utilizing the 'to_regoperator' function
```

```sql
SELECT opr.oid, opr.oprname, opr.oprcode, ppr.oprresult,
       opr.oprleft  :: oid :: regtype,
       opr.oprright :: oid :: regtype
FROM   pg_operator AS opr
WHERE  opr.oid = to_regoperator('op(τ_l,τ_r)') :: oid;
```

Listing 3.12: Several approaches on retrieving operators

An example of such operator overloading would be `+` on the operands `date` and `integer` in Table 3.9. From the operator alone a user is not able to infer the function powering it. The SQL function call of `to_regoperator` would be made using the string `'+(date,integer)'`.

```sql
SELECT ('1998-11-09' :: date) + 1 AS "Date of Birth";
```

```
 Date of Birth
---------------
 1998-11-10
(1 row)
```

Table 3.9: Overloaded Operator + in SQL

Where the above-mentioned approach to finding suitable functions is based on lookups on the System Catalogs, those gain their information from the definition of the types and their function definitions in the source code. Hence a more tedious path to pursue can be searching the source files for functions that should be used in the translated source code. This path yields documentation for the functions that may be missing from the official end-user documentation.

Talking about source code. Until now this Section has been about finding out the name or OID of a function or operation. Yet not its translated call. These functions are accessible through the same header files as the types themselves. The functions exposed there are bound by the same *version 1* calling conventions as our function is. They are *Magic Functions* themselves. Therefore we cannot translate a PL/pgSQL function or operator call into $v_{res} = f(v_1, \dots, v_n)$; and call it a day.

When interacting with a *Magic Function* we do so by issuing a `DirectFunctionCall`. The arguments of the function call are the original function we want to call as well as its arguments as of type `Datum`. These original n-ary functions are called with a corresponding $n+1$-ary `DirectFunctionCall`n or `DirectFunctionCallOid`n to be able to hand over both the function (or OID) as well as its arguments. In Table 3.10 these cases are shown as well as the case that a function name is overloaded and additional markers, such as the argument types, are needed. This case also applies for PL/pgSQLUDFs.

Another option would be to set up a query of the likes of `SELECT` $f(v_1, \dots, v_n)$; and execute it as seen in Section 3.4.

3.8 Arrays

Composite Types or Row Types present a way to bundle information. Arrays allow for it as well. They are a container data structure present in the SQL standard since SQL:1999 [13]. PostgreSQL

PL/SQL	C
	// DirectFunctionCall$n$
	Datum v_{res} = DirectFunctionCall$n$(f, v_1, …, v_n);
	// DirectFunctionCallOid$n$ - unique name
	Oid oid_f = DatumGetObjectId(
	DirectFunctionCall1(to_regproc, CStringGetDatum('f'));
f(v_1, …, v_n)	Datum v_{res} = DirectFunctionCallOid$n$(oid_f, v_1, …, v_n);
	// DirectFunctionCallOid$n$ - overloaded name
	Oid oid_f = DatumGetObjectId(
	DirectFunctionCall1(to_regprocedure,
	CStringGetDatum('f(τ_1, …, τ_n)'));
	Datum v_{res} = DirectFunctionCallOid$n$(oid_f, v_1, …, v_n);

Table 3.10: Translated state of function calls.

implements a plethora of operations [14] on this type of homogenous collection of values. This Section should provide several approaches working with PostgreSQL Arrays.

In SQL there are several ways to construct Arrays. The most straightforward is to explicitly state which values or identifiers are put into the Array.

$$\texttt{ARRAY}[v_1,\dots,v_n]$$

Where the Array type can not be inferred from the context a user may explicitly annotate or cast the array to an explicit type $\_\tau$.

$$\texttt{ARRAY}[v_1,\dots,v_n] :: \tau[] \quad \text{or} \quad \texttt{ARRAY}[v_1,\dots,v_n] :: \_\tau$$

Other than that a query may return an Array or it can be constructed by Array operations. As all Array Types have overlapping functionality like the aforementioned plethora of operations there exists a more generalized type called `anyarray`. This allows for functions that are not dependent on the type of element stored in the Array to be implemented in a more abstract approach.

Besides this more general type Arrays are more seen as an extension to existing types. Every Type has a corresponding Array Type. The OIDs of said Array Types are the `typarray` column in the System Catalog `pg_type`. Also these OIDs appear as their own Type in `pg_type`. There the column `typname` presents itself in the style of $\_\tau_{original}$.

This explains the concept of these Arrays. The next step would be to work with them in C. Starting one could assume that the PostgreSQL Array is a C Array in disguise or a C struct with a C Array and more meta information. That would be false. The PostgreSQL Array internally is of `ArrayType`. A struct with fields for the length, dimensions, element type and data offset.

```
ArrayType at =
  DatumGetArrayType($v_{input}$);

Datum *$v_{output}$;
bool  *$v_{output}$_NULLs;
int    $v_{output}$_cardinality;

int16 elmlen;
bool  elmbyval;
char  elmalign;

get_typlenbyvalalign(
   $v_{input}$->elemtype,
  &elmlen,
  &elmbyval,
  &elmalign);

deconstruct_array(
   $v_{input}$,
   $v_{input}$->elemtype,
   elmlen,
   elmbyval,
   elmalign,
  &$v_{output}$,
  &$v_{output}$_NULLs,
  &$v_{output}$_cardinality);
```

Deconstruction

```
int16 elmlen;
bool  elmbyval;
char  elmalign;

get_typlenbyvalalign(
   $\tau_{element}$,
  &elmlen,
  &elmbyval,
  &elmalign);

Datum $v_{output}$ =
  PointerGetDatum(
    construct_array(
      $v_{inputs}$,
      $v_{inputs}$_cardinality,
      $\tau_{element}$,
      elmlen,
      elmbyval,
      elmalign));
```

Construction

Table 3.11: PostgreSQL Array deconstruction to and construction from C

3.8.1 PostgreSQL Array → C Array → PostgreSQL Array

A first approach knowing that the PostgreSQL Array is not a "normal" Array could be to find a way to convert it into a C Array working on it and then converting it back into a PostgreSQL Array. The procedure `deconstruct_array` does exactly what its name suggests. The left Listing in Table 3.11 shows that given the input `ArrayType` and element information a `Datum` Array can be filled. Including a corresponding Nulls Array as well the length of both Arrays.

The inverse operation can be seen on the right side of said table. Given a `Datum` Array v_{inputs}, its elements OID and cardinality can again be used to retrieve element information and construct a new `ArrayType`.

This approach seems sensible for small Arrays but regarding typical SQL operations such as concatination and pre- or appending are not as comfortable as in SQL. Here the power lies with `DirectFunctionCalls` as seen in Section 3.7. For that the Arrays have to be present as `Datums`. When searching for compatible functions use $\tau \equiv$ `anycompatiblearray` in Listings 3.10 and 3.11.

4 Problems

On the path of implementing this work, we encountered a few topics that may seem trivial afterward but still presented themselves as sufficient road blocks or became of surprising significance. Take these as a lesson for ones own start of developing C language PostgreSQL functions or extension.

4.1 Interaction with Extension Creation

Extensions are applied to a database with the `CREATE EXTENSION` DDL statement. This is similar to other DDL statements that create tables, types or indices. Where those are direct definitions of such, extensions are collections of a multitude of these statements. Having multiple definitions round-up like this occludes the complexity of the extension.

The UDFs placed there may be dynamically loaded – inferring OIDs and other information at runtime – but their own argument or result types fall into the same type system as everything else. Modifying a dependent type with `CREATE OR REPLACE` or `DROP …; CREATE …;` affects these UDFs.

To relieve the extension of its dependencies move those into the extensions as well.

4.2 PostgreSQL Architecture and Source Code

PostgreSQL documents the user-side functionality very well. Also, introductory documentation for developing extensions is provided. [8] This paradigm of easy onboarding shifts a little bit when investigating source functionality. There is no overview of the architecture other than the source code itself.

The `Datum`-only approach allows for slight ease of use. To identify constructs like `array_iter` or `ExpandedRecordHeader` that are crucial for their respective cause can not be found by querying the System Catalogs but rather have to be found in the source code.

To scour it by opening all files and searching for similar usage is one option. Another is using the *doxygen*[1] platform hosted by the organization behind PostgreSQL. It acts as a reference with the latest source version. There one can follow types and functions to their definitions and see where they are referenced and used.

[1]https://doxygen.postgresql.org/index.html

4.3 Different Release Versions

Different PostgreSQL release versions have a different code base. So far so obvious. This results in the C extension being compatible with the version it has been compiled for. Compared to ”normal” PL/SQL language extensions or scripts it is therefore bound to a specific database version. A database version upgrade would not affect the SQL scripts. The PostgreSQL functionality used in C language functions might be removed, renamed or replaced entirely and could not work in a different database version.

Coinciding with these changes the performance of the extension (or all queries for that matter) might be impacted. Where previously one expected a certain behavior from the internals, there could have been a more performant replacement parallel to the expected piece of code.

4.4 Interfacing with Arrays

In the Section ’A Critique of the POSTGRES Data Model’ of the 1990 Paper ’The implementation of POSTGRES’ by [15] they said that they felt making mistakes when implementing arrays. They even conclude that ”In retrospect, we should have included general support for arrays or no support at all” [15].

Since then there were both variable- and fixed-length arrays [16] where the initial SQL specification introducing arrays (SQL:1999 [13]) nine years later solely expected fixed-length arrays. Variable length arrays were later introduced into the specification in SQL:2003. During this evolution of standards, the POSTGRES Project evolved itself into PostgreSQL. Since then the concepts of having both types of arrays did not change. This results in a confusing number of functions related to arrays.

Additionally, arrays can be embedded into each other, Multidimensionality does not pose a problem per sé but as the internal representation keeps them one dimensional and adds another struct type wrapping them leading to more functions involved in working with them.

Together with arrays being a universal concept across different programming languages but being slightly different in SQL and more different in the backend can end in a slight disconnect.

5

Meassurements

We translated sixteen PL/pgSQL UDFs applying the rules described in Chapter 3. These manually-translated UDFs were benchmarked in PostgreSQL 14.2 on a 64 Bit Linux platform comprised of 2× AMD EPYC® 7402 CPUs at 2.8 GHz and 2 TB of RAM.

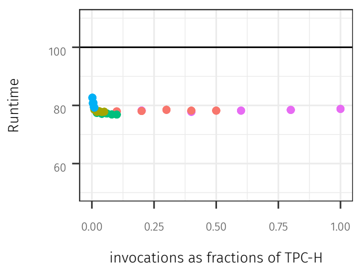
The original UDFs were implementations of several TPC-h benchmark queries [17] in PL/pgSQL. The C translations of these UDFs are therefore also viable implementations of these queries. The background of the TPC-h benchmark is to provide real-world database schemata as well as queries applicable to them. The sizes of the databases behind them also are sized according to a *Scaling Factor*.

Additionally to these real-world examples, there are some academic use cases like a bounding box, n-body simulation or marching square algorithm.

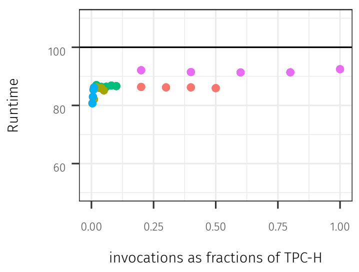
These 16 different UDFs are of different complexity. The complexity consists of the number of embedded queries ($|Q_i|$), the number of *loop constructs* (LC) and their respective embedding into each other as well as their *Cyclic Complexity* (CC). They also work on different types, from built-ins to composites to user-defined composite types. Those are not only arguments or return types but internal variables as well. Table 5.1 presents these complexity measures and the performance of the 16 C language UDFs compared to the original PL/pgSQL variants.

Queries that show a slightly better improvement than the other general improvement like `late` or `service` are of simpler complexity. All embedded queries are executed before entering decisions or loops. The types used in it are also compatible with C primitives leading to a nigh perfect embedding of the control flow within C.

Where the previous TPC-H examples had improved runtime more complex UDFs like `bbox`



(a) Runtime for UDF `late`



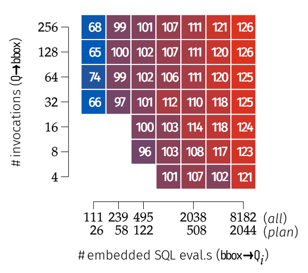
(b) Runtime for UDF `service`

Figure 5.1: Runtime comparison for the different scaling factors (001●, 005●, 01●, 05●, 1●)

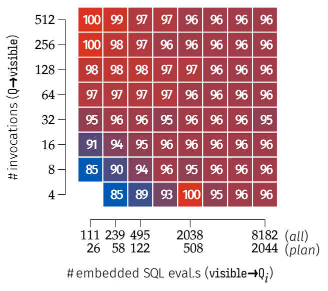
	UDF	Return Type	$\lvert Q_i \rvert$	LC	CC	Runtime (speedup)
`bbox`	detect bounding box of a 2D object	`box`	2	1	5	111.04% (0.90×)
`force`	n-body simulation (Barnes-Hut quad tree)	`point`	3	1	5	190.53% (0.52×)
`global`	does a TPC-H order ship intercontinental?	`boolean`	1	1	3	90.23% (1.11×)
`items`	count items in hierarchy (adapted from [18])	`int`	2	1	2	19.75% (5.06×)
`late`	find delayed orders (transcribed TPC-H Q 21)	`boolean`	1	1	4	77.24% (1.29×)
`march`	track border of 2D object (Marching Squares)	`point[]`	2	1	5	110.26% (0.91×)
`march-tvf`	track border of 2D object (Marching Squares)	`point[]`	2	1	5	102.65% (0.97×)
`margin`	buy/sell TPC-H orders to maximize margin	`row`	3	1	5	98.62% (1.01×)
`markov`	Markov-chain based robot control	`int`	3	1	3	94.48% (1.06×)
`savings`	optimize supply chain of a TPC-H order	`row`	6	1	4	83.72% (1.19×)
`sched`	schedule production of TPC-H lineitems	`row array`	5	2	6	99.98% (1.00×)
`service`	determine service level (taken from [19])	`text`	1	0	3	86.43% (1.16×)
`ship`	customer's preferred shipping mode	`text`	3	0	3	95.58% (1.05×)
`sight`	compute polygon seen by point light source	`polygon`	3	2	3	97.86% (1.02×)
`visible`	derive visibility in a 3D hilly landscape	`boolean`	2	1	3	96.67% (1.03×)

LC ≡ Loop Constructs, CC ≡ Cyclic Complexity

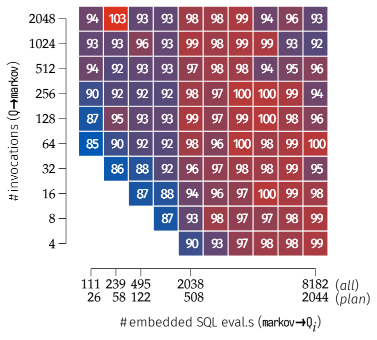
Table 5.1: A collection of PL/SQL UDFs with compared to a C implementation said UDFs



(a) Heat map for UDF `bbox`.



(b) Heat map for UDF `visible`.



(c) Heat map for UDF `markov`.

Figure 5.2: Runtime for context switches varied (lower/lighter is better).

(Figure 5.2a did not show such positive development. As the function works on composite and custom data types a na"ive implementation did not suffice. We see the problem in the decomposition of said composite types. Having to deconstruct and reassign values seems to be the culprit.

Other academic examples like `visible` (Figure 5.2b) seem to have to gain slightly from a C language implementation.

6 Conclusion and Related Work

6.1 Conclusion

In this work, we attempted to translate several UDFs from their PL/pgSQL definition into an C language analogue. To achieve this part we inferred several rules to translate domain-specific constructs into C. Coinciding with the need to introduce PostgreSQL specific types, constructs and conventions something is lost from the simple PL/pgSQL implementation. This something is the clarity of the written code. The boilerplate code needed to achieve a successful translation using the same constructs the PL/pgSQL interpretation would use obscures the control flow and intend the original query communicates clearly.

Besides the aspect of translating the benchmark UDFs we ran them against their original variants. The experiment identified a minimal to no overhead imposed by PostgreSQL's PL/pgSQL interpreter.

6.2 Future and Related Work

This work only touches the functionality provided by PostgreSQL. Future work might want to take a deeper dive into PostgreSQL's architecture and prove a more concise or efficient implementation. In this work, we encountered Memory Contexts but did not use them to their full extent. The same is true for Arrays. It would be interesting whether a more well-rounded solution has an edge compared to the original versions.

Also we only have taken a look at PostgreSQL. It could be of interest whether other Database Management Systems handle their PL/SQL implementation differently. Better or worse. Such systems could be direct competitors or newer systems like DuckDB.

Besides the works [1], [2] and [3] cited in the Introduction there is the work [20] on the ByePy Compiler that follows the same approach. The difference being the host language having embedded queries is Python. Based upon the rule set identified in our work one could attempt to compile PL/pgSQL directly into a C extension and therefore automating what has been done here by hand.

Listings

Bibliography

[1] Christian Duta, Denis Hirn, and Torsten Grust. *Compiling PL/SQL Away*. Sept. 7, 2019. arXiv: 1909.03291[cs]. URL: http://arxiv.org/abs/1909.03291 (visited on 01/27/2023).

[2] Denis Hirn and Torsten Grust. "PL/SQL Without the PL". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 2677–2680. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3384678. URL: https://doi.org/10.1145/3318464.3384678 (visited on 10/28/2022).

[3] Denis Hirn and Torsten Grust. "One WITH RECURSIVE is Worth Many GOTOs". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, June 18, 2021, pp. 723–735. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3457272. URL: https://doi.org/10.1145/3448016.3457272 (visited on 10/28/2022).

[4] *PostgreSQL: Documentation: 14: PostgreSQL 14.7 Documentation*. URL: https://www.postgresql.org/docs/14/index.html (visited on 03/18/2023).

[5] *PostgreSQL: Documentation: 14: PL/pgSQL — SQL Procedural Language*. URL: https://www.postgresql.org/docs/14/plpgsql.html (visited on 03/18/2023).

[6] *PLpgSQL Release in Postgres 6.4*. PostgreSQL Documentation. Jan. 1, 2012. URL: https://www.postgresql.org/docs/7.2/release-6-4.html (visited on 01/10/2023).

[7] *PostgreSQL: Documentation: 14: System Catalogs*. URL: https://www.postgresql.org/docs/14/catalogs.html (visited on 03/18/2023).

[8] *PostgreSQL: Documentation: 14: C-Language Functions*. URL: https://www.postgresql.org/docs/14/xfunc-c.html (visited on 03/18/2023).

[9] *PostgreSQL: Documentation: 14: Control Structures*. URL: https://www.postgresql.org/docs/14/plpgsql-control-structures.html#PLPGSQL-CONTROL-STRUCTURES-LOOPS (visited on 03/18/2023).

[10] *PostgreSQL: Documentation: 14: Monetary Types*. URL: https://www.postgresql.org/docs/14/datatype-money.html (visited on 03/18/2023).

[11] *PostgreSQL: Documentation: 14: Geometric Types*. URL: https://www.postgresql.org/docs/14/datatype-geometric.html (visited on 03/18/2023).

[12] *PostgreSQL: Documentation: 14: Date/Time Types*. URL: https://www.postgresql.org/docs/14/datatype-datetime.html (visited on 03/18/2023).

[13] Andrew Eisenberg and Jim Melton. "SQL: 1999, formerly known as SQL3". In: *ACM SIGMOD Record* 28.1 (Mar. 1999), pp. 131–138. ISSN: 0163-5808. DOI: 10.1145/309844.310075. URL: https://dl.acm.org/doi/10.1145/309844.310075 (visited on 03/14/2023).

[14] *PostgreSQL: Documentation: 14: Arrays*. URL: https://www.postgresql.org/docs/14/arrays.html (visited on 03/18/2023).

[15] M. Stonebraker, L.A. Rowe, and M. Hirohama. “The implementation of POSTGRES”. In: *IEEE Transactions on Knowledge and Data Engineering* 2.1 (Mar. 1990), pp. 125–142. ISSN: 10414347. DOI: 10.1109/69.50912. URL: http://ieeexplore.ieee.org/document/50912/ (visited on 10/28/2022).

[16] Lawrence A. Rowe and Michael Stonebraker. “The POSTGRES Data Model”. In: *Proceedings of the 13th International Conference on Very Large Data Bases*. VLDB ’87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1, 1987, pp. 83–96. ISBN: 978-0-934613-46-0. (Visited on 10/28/2022).

[17] *TPC-H Homepage*. URL: https://www.tpc.org/tpch/ (visited on 03/18/2023).

[18] Ravindra Guravannavar and S. Sudarshan. “Rewriting procedures for batched bindings”. In: *PVLDB* 1 (Aug. 2008), pp. 1107–1123. DOI: 10.14778/1453856.1453975.

[19] V. Simhadri K. Ramachandra A. Chaitanya R. Guravannavar S. Sudarshan. “Decorrelaion of User Defined Functions in Queries”. In: ICDE. Chicago, IL, USA, Mar. 2014.

[20] Tim Fischer, Denis Hirn, and Torsten Grust. “Snakes on a Plan: Compiling Python Functions into Plain SQL Queries”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. New York, NY, USA: Association for Computing Machinery, June 11, 2022, pp. 2389–2392. ISBN: 978-1-4503-9249-5. DOI: 10.1145/3514221.3520175. URL: https://doi.org/10.1145/3514221.3520175 (visited on 10/28/2022).