Data is Data and Control Should be Data, Too

Compiling Iterative Table-valued PL/SQL UDFs into Recursive SQL Code

Denis Hirn\textsuperscript{1}
\vspace{0.5cm}
\textsuperscript{1}supervised by Torsten Grust
University of Tübingen, Germany

\begin{abstract}

PL/SQL functions suffer from poor runtime performance due to the frequent context switches that occur between the PL/SQL interpreter and the SQL executor. This switching causes friction that can slow down UDF execution significantly. Table-valued UDFs incur the additional challenge of the efficient treatment of the sizable results they generate. In this paper, we generalize our PL/SQL UDF compilation strategy to also handle such table-valued UDFs. The generated SQL code carefully separates control flow from data flow at runtime. Compiled UDFs efficiently stream their table-valued results (as opposed to UDF variants that need to hold and copy intermediate states in array variables) and thus impose significantly less memory pressure.
\end{abstract}

\section{Introduction}

PL/SQL is a high-level procedural programming language that allows developers to write custom user-defined functions (UDFs), operators, and algorithms that are not supported by the built-in functions of the database system. PL/SQL enables a style of imperative programming—which is quite different from the declarative set-oriented SQL paradigm—but still provides immediate access to database-resident tables. The distinctive PL/SQL features are (1) destructive variable assignments, (2) statement sequences, (3) arbitrary control flow (e.g., in terms of IF...ELSE, WHILE, EXIT), and a (4) seamless integration of SQL queries and expressions.

Typically, PL/SQL is implemented as an interpreted language on top of SQL host engines. The imperative, non-SQL statements are interpreted by the PL/SQL subsystem, while all embedded SQL queries are sent to the SQL executor. Because both execution environments are completely disparate, each switch from one context to the other (and back) takes time and therefore causes context switching overhead. The situation is particularly dire when these embedded queries are placed in tight PL/SQL (FOR or WHILE) loops. In that case, switching back and forth between PL/SQL and the SQL executor occurs very frequently, which multiplies the overhead, and ultimately slows down execution. Figure 1 visualizes this.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure1.png}
\caption{Context switching as UDF \texttt{march} executes.}
\end{figure}

It has thus become common developer lore that PL/SQL is slow and should be avoided if possible [1]. It has nonetheless been used for decades to implement complex database-driven applications [2]. Research has since recognized this as an important issue, resulting in several publications addressing this pressing challenge [1, 3–5].

The scope of the PhD project. The overarching goal of the PhD project is to allow developers to use imperative programming, e.g., in the form of PL/SQL UDFs, while maintaining the high performance of plan-based SQL execution. To accomplish this, we develop new ways to compile imperative PL/SQL UDFs into SQL queries. This compilation can handle arbitrary nesting of the PL/SQL features mentioned on the left. This includes looping control flow. As a side effect of this effort, database systems without PL/SQL support—but with support for a contemporary SQL dialect—will be able to run imperative PL/SQL UDFs after compilation, since no PL/SQL interpreter is required. In the present paper, we focus on the compilation of imperative (typically: iterative) table-valued UDF code into recursive yet plain SQL queries.

\subsection{From Scalar Values to Tables}

In [5], we described a compiler that transforms \textit{scalar} PL/SQL UDFs to a single recursive SQL CTE (WITH RECURSIVE). While keeping the basic idea and compilation chain as is, the present work separates the management of control flow and data flow to make the compilation suitable for \textit{table-valued} UDFs. To this end, we introduce the concept of \textit{control rows} and \textit{data rows}. Previously, the compiler used only control rows and could not handle table-valued UDFs.

Let us look at an example. UDF \texttt{march} of Figure 2a is a
Table-valued version of PL/SQL UDF march.

Table-Valued version of PL/SQL UDF (a) march(vec2)

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).

Table-Valued version of PL/SQL UDF march.

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).

Table-Valued version of PL/SQL UDF march.

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).

Table-Valued version of PL/SQL UDF march.

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).

Table-Valued version of PL/SQL UDF march.

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).

Table-Valued version of PL/SQL UDF march.

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).

Table-Valued version of PL/SQL UDF march.

Marching Squares as an array-based PL/SQL UDF. (b) march-arr(vec2).
We call these rows "control rows." The recursive part of run in Lines 4 to 23 of Figure 5 implements a dispatcher. Figure 6 depicts the central role of the dispatcher "trampoline" and how it realizes the control flow for UDF march. The SQL query reads the call column to select one block ∈ \{while, exit\} for evaluation. All blocks must return a new control row with columns "rec?,data?" and call, so the dispatcher knows how to proceed in the next iteration (see Lines 13, 16, and 21 of Figure 5). This process continues until a block returns a control row with column "rec?"=false (see Line 21 of Figure 5). The working table in the next iteration will be empty, and \texttt{WITH RECURSIVE} evaluation stops.

**New: Data Flow Management.** While scalar UDFs return a single value in the last trampoline iteration, table-valued UDFs can return any number of values during execution (see \texttt{emit cur0} in Figure 4). The CTE of Figure 5 encodes these returned values in dedicated rows marked \( \square \) in Line 8 of Figure 5. Two columns manage this data flow:

data? \in \{true, false\}: Column data? indicates if this row has a valid return value in column res.

res: Contains this return value.

We call rows with column data?=true \texttt{data rows}. When a UDF uses either \texttt{RETURN NEXT} or \texttt{RETURN QUERY}, such data rows are created in addition to control rows.

Given the UDF of Figure 2a and assuming a call \( \text{march}(8,7) \), overall the recursive CTE computes table run as shown on the next page. After the initialization, marked \( \circ \), each iteration (separated by \( \cdots \)) generates two rows, a data row and a control row. (In general, any number of data rows can be created in each iteration.) Note how the last iteration indicates the end of execution via (rec?,data?)=(false,false).
Recall that the original PL/SQL UDF has to materialize its table-valued result during execution, and returns all of it as a whole. This materialization prevents the surrounding execution plan from terminating prematurely, for example, when a LIMIT clause is used: 

```sql
SELECT * FROM march((8,7)) LIMIT 5
```

After compilation, however, these results are immediately returned to the parent operator in terms of data rows, without materialization of the entire result. This saves memory and reduces the runtime. In addition, metrics such as CPU cost and cardinalities can be estimated more accurately, making planning of the translation more effective: While PL/SQL UDFs are effectively a black box for the planner, the translation is a regular SQL query that the planner is designed to handle.

### 3. Data Rows in Trampolined Style

Both UDFs, `march` and `march-arr`, indeed exhibit the infamous context switching overhead that gives PL/SQL its bad reputation. We have measured that the back and forth between PL/SQL and SQL accounts for 20% of the overall evaluation time for both variants (see Table 1). The compilation to recursive SQL CTEs described in [5] avoids this particular overhead for the two UDFs.

However, the naive treatment of the iterative `result` array construction and copying in the CTE for the scalar UDF `march-arr` quickly eats up all the gains: the quadratic array maintenance cost mentioned in the introduction add up to about 50% of the overall CTE runtime. If we double the size of UDF input, the working table of the CTE for `march-arr` grows by a factor of four (from 16 MB to 64 MB) and the array maintenance overhead increases to 56%. Ultimately, this leads to a slowdown of `march-arr` after compilation.

In stark contrast, the control- and data-flow-aware compilation strategy sketched in Section 2, translates the table-valued UDF into the recursive SQL CTE of Figure 5.

<table>
<thead>
<tr>
<th>Type</th>
<th>Overhead</th>
<th>Runtime</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>march-arr vec2</td>
<td>20%</td>
<td>112.8% (0.88x)</td>
<td>16 MB</td>
</tr>
<tr>
<td>march SETOF vec2</td>
<td>20%</td>
<td>38.2% (2.61x)</td>
<td>110 kB</td>
</tr>
</tbody>
</table>

### 4. Wrap-Up

Separating the concepts of data rows and control rows is essential for translating table-valued functions. We save working table space and are rewarded with a significant runtime advantage over array-centric UDF alternatives. But it does not stop there. In the future, we plan to generalize this concept to add support for recursion in UDFs. Data rows could be used to model call stack entries, which—in one form or another—are required for functions that are not tail-recursive.

A further generalization would be to remove the restriction that one control row always yields exactly one new control row. This property causes trampolined style to model single-threaded computation. If the SQL backend supports parallel plan execution, the creation of multiple control rows in a single iteration effectively spawns independent threads. UDF evaluation would benefit from parallelization just like regular SQL queries.

### References


