EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Masterthesis Media Informatics

From Recursion To Iteration: Compiling SQL UDFs with Continuations

Tobias Burghardt

30. September 2021

Examiner

Prof. Dr. Torsten Grust

Co-Examiner

Prof. Dr. Klaus Ostermann

Supervisor

Denis Hirn

Burghardt, Tobias:
From Recursion To Iteration: Compiling SQL UDFs with Continuations
Masterthesis Media Informatics
Eberhard Karls Universität Tübingen
From May 01, 2021 to September 30, 2021

Abstract

In Structured Query Language (SQL) a User-defined Function (UDF) can be defined recursively. Due to the plan-based evaluation of SQL every recursive call inside a UDF's body is painful at runtime. Transformation techniques such as Continuation-Passing Style (CPS), defunctionalization and trampolined style are widely known in the programming language community and can be employed to accommodate the evaluation strategy of SQL.

This thesis proposes a SQL-to-SQL compiler to translate a recursive UDF into a recursive Common Table Expression (CTE) that can be evaluated efficiently by the Relational Database Management System (RDBMS).

Additional optimizations (foremost memoization) can further trim down the runtime of the recursive CTE.

Contents

ACRONYMS

SQL	Structured Query Language
UDF	User-defined Function
RDBMS	Relational Database Management System
CTE	Common Table Expression
SSA	Static Single Assignment
ANF	Administrative Normal Form
CPS	Continuation-Passing Style

1 INTRODUCTION

1.1 Problem

Recursion is a highly appreciated problem-solving strategy in programming. However, solving complex recursive problems through recursive CTEs can cause a hard time. This can drive programmers to utilize a programming language other than SQL for their complex computations on the data. In this case, the data must be transferred with additional effort whereby the commandment of *moving the computation close to the data* [1] is hurt and any support of the RDBMS is lost.
A second alternative would be to use recursively defined UDFs — let us call them *functional-style UDFs*. Using functional-style UDFs programmers can write their readable and compact recursive functions just as they would do in many other programming languages. But a major drawback is that RDBMSs do not endorse the use of functional-style UDFs. More precisely, unoptimized functional-style UDFs result in poor runtime performances.

For some popular RDBMSs, these are the reasons that lead to problems with functional-style UDFs [2]:

- **PostgreSQL**: The plan-based evaluation of functional-style UDFs is SQL's doom. Instead of parsing, analyzing, and planning the body of the UDF once, this process happens for every recursive call. Even when the plan is cached, it still has to be reinstantiated and teared down every time. Therefore, the actual useful work to evaluate the UDF often accounts just for a small fraction of the overall runtime. The majority of the runtime is spent for parsing, analyzing and planning.
- **Microsoft SQL Server** and **Oracle**: Maximum UDF recursion depth of 32/50.
- **MySQL** and **HyPer**: General prohibition of functional-style UDFs.
- **SQLite3**: No support of UDFs in general.

1.2 Goal

While plan-based evaluation may be appropriate for queries, it is not at all for functional-style UDFs. Therefore, functional-style UDFs should be treated as what they are, namely functions and not queries. This means that the repeated process of parsing, analyzing and planning a UDF's body after each recursive call is to be avoided.
The goal of this thesis is to transform a functional-style UDF into an equivalent SQL:1999 recursive CTE. During this transformation the recursion inside the functional-style UDF is removed whereby the UDF's body has to be parsed, analyzed and planned only once. That way the runtime is mainly determined by the actual evaluation time and not on repetitive overhead.
As a consequence, the transformed recursive CTE is more efficient than the functional-style UDF and less restricted by RDBMSs.

This thesis is partitioned into 6 chapters starting with the introduction in this chapter 1. In chapter 2 the principles of recursive CTEs are explained and it is described how related work realized their transformation approaches into recursive CTEs. Chapter 3 presents the compilation approach of this thesis and chapter 4 proposes optimizations that can further improve this approach. The experiments in chapter 5 emphasize the positive runtime effect and the last chapter 6 gives a brief summary together with an outlook.

2 BACKGROUND

This chapter gives an overview on how recursive CTEs are implemented. This is important in order to understand why a recursion-removal approach is necessary to express a recursive problem with a recursive CTE.
Apart from that, this chapter presents related work that also compile (functional-style resp. PL/SQL) UDFs into recursive CTEs.

2.1 Recursive Common Table Expressions

SQL indicates recursive CTEs by a `WITH RECURSIVE` construct (see Figure 2.1). A recursive CTE is divided into two parts which are combined using `UNION` or `UNION ALL`. One part is a *non-recursive term* q_0 and the other part is a *recursive term* q_{rec} that can reference to the recursive CTE.

```
WITH RECURSIVE
  <T>(<c1>, ..., <ck>) AS (
   <q0>                          -- non-recursive term
      UNION [ALL]                -- either UNION or UNION ALL
   <qrec(T)>                      -- recursive term refers to T
  )
<q>(<T>)                          -- post-processing
```

Figure 2.1: Common schema of a recursive CTE.

The procedure works as follows [3]:

- q_0 is evaluated once and the result (in case of `UNION`: after removing duplicates) is added to result table `T` and to a *working table*.
- q_{rec} is evaluated (repeatedly) until a produced working table is empty:
  1. The self-reference of `T` is replaced by the rows of the current working table.
  2. q_{rec} is evaluated (in case of `UNION`: by removing rows that are already contained in `T`) and the rows are added to `T` as well as to an *intermediate table*.

3. The rows of the working table are replaced by the rows of the intermediate table and all rows inside the intermediate table are removed.

- If the new working table is empty `T` is left to the query `q` in the postprocessing. Otherwise q_{rec} is evaluated again in the same manner.

  "Strictly speaking, this process is iteration not recursion, but `RECURSIVE` is the terminology chosen by the SQL standards committee." [3]

This citation is taken from the *PostgreSQL* documentation. So if a functional-style UDF shall be transformed into a recursive CTE, the recursion has to be removed and replaced by a loop construct.

2.2 Related Work

2.2.1 Functional-Style SQL UDFs With a Capital 'F'

The paper *Functional-Style SQL UDFs With a Capital 'F'* [2] proposes one way to compile functional-style UDFs into their efficient pendants in terms of using a recursive CTE.

This SQL-to-SQL compilation proceeds in two steps:

1. Call Graph Construction
   - When a functional-style UDF is invoked with arguments `args` the corresponding call graph will be constructed.
   - Starting with `args` as root and ending with the base cases as leaves, all resulting argument combinations of the recursive calls (that would have to be performed but aren't yet) are attached to the appropriate place in the call graph.
2. Bottom-Up Evaluation
   - Evaluation starts with the arguments in the leaves of the call graph and ends with the root.

During the evaluation all intermediate results are stored into a *working table*. This holds potential for a possible *memoization* i.e. the storing and reusing of already computed results [1]. By remembering intermediate results they act like base cases in

[1] We will explicitly analyze this memoization optimization in section 5.3.

the call graph and therefore do not attach further nodes to it. That way the size of the call graph reduces and less work has to be done.
However, this is not the only optimization discussed in the paper. There are other optimizations like *reference counting* and the *exploitation of linear and tail recursion*.

Reference counting drops return values from the working table if it is ensured that they aren't needed any longer for the computations of further results. Thus, the working table becomes smaller and with it the runtime decreases.
For linear- and tail recursive functions the call graph construction and evaluation can be simplified [2]. Details can be studied in the paper itself.

2.2.2 One WITH RECURSIVE is Worth Many GOTOs

SQL is a declarative programming language. However, PL/SQL adds the possibility to write programs in an imperative style.
It supports [4]:

- stateful variables
- complex and looping control flow (e.g., `IF ... ELSE`, `LOOP`, `WHILE`, `FOR`, `EXIT`, or `CONTINUE`)
- the embedding of SQL queries inside PL/SQL

The use of SQL queries inside PL/SQL produces context switches between SQL and PL/SQL. The overhead that these context switches produce is very time-consuming. To avoid context switches a compilation of PL/SQL UDFs into recursive CTEs is desirable.

The paper *One WITH RECURSIVE is Worth Many GOTOs* [4] achieves this in a 4-stage-compilation:

1. PL/SQL to Static Single Assignment (SSA)
    - Reduction of the complexity by transforming arbitrary iterative control flow into `GOTO`-based control flow.
2. SSA to Administrative Normal Form (ANF)
    - This stage turns the imperative program into a functional program.
    - Each basic block gets turned into a function:
        - Assignment statements are expressed as `LET` bindings.
        - `GOTO` is replaced by a tail call to the corresponding function.

    - ⇒ That way all functions become tail-recursive and mutually call each other.
  - If possible, Inlining is used to reduce the number of functions.
3. Tail Recursion to Trampolined Style
  - Elimination of the mutually tail-recursive functions to match the single-cycle iteration scheme of a recursive CTE.
  - This transformation returns a function `start` which repeatedly executes a dispatcher function `trampoline` until the overall result is calculated.
4. Trampolined Style to recursive CTE
  - All function bodies are translated into SQL-`SELECT` blocks.
  - These blocks are fitted into a `WITH RECURSIVE`-based SQL template:
    - The `SELECT` block of `start` builds the non-recursive term.
    - The other `SELECT` blocks are combined through `UNION ALL` to build the non-recursive term.

Similar transformation steps will also be considered in the course of this thesis. For example, the transformation into ANF produces tail recursive functions just like the *CPS-Transformation* that is used and explained in chapter 3. In addition, the *trampolined style* will also play an important role there.

3 TRANSFORMATIONS

This chapter explains the SQL-to-SQL compilation of functional-style UDFs into recursive CTEs. Thereby the recursion is converted step by step into an iteration.

To practically demonstrate the transformation steps, the following sections show how to apply them to a simple and compact UDF `fib(n)`. It calculates the n-th number in the *fibonacci sequence*.
The fibonacci sequence [5] is infinite and defined on the natural numbers $\mathbb{N}$. The first two fibonacci numbers are predefined ($fib(1) = fib(2) = 1$ [1]) and subsequent fibonacci numbers are defined as the sum of its two direct predecessors in the fibonacci sequence (see Figure 3.1).

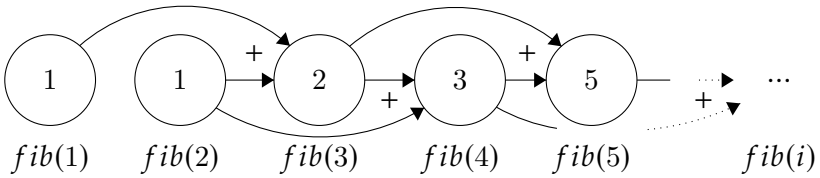

Figure 3.1: Principle of the fibonacci sequence.

A recursive, textbook style formulation of the fibonacci sequence is depicted in Figure 3.2:

$$fib(n) = \begin{cases} 1 & n \in \{1, 2\} \\ fib(n-1) + fib(n-2) & \forall n > 2, n \in \mathbb{N} \end{cases}$$

Figure 3.2: Algorithm in textbook style.

[1] Sometimes you will also find $fib(0) = 0$.

3.1 Plain Function

The functional-style UDF `fib(n)` can be derived very easily from the textbook style (see Figure 3.2). In Figure 3.3, you see a derivation of it [2].

```
CREATE FUNCTION fib(n numeric) RETURNS numeric AS
$$
SELECT 1
WHERE n = 1 OR n = 2
    UNION ALL
SELECT fib(n-1) + fib(n-2)
WHERE n > 2
$$ LANGUAGE SQL STABLE STRICT;
```

Figure 3.3: functional-style UDF of fibonacci sequence.

Some of the transformation steps produce higher-order funtions which are not expressible in raw SQL. Thus, a switch to a functional programming (FP) language (here: *Haskell*) is necessary to practically perform the transformation steps. After the UDF is transformed into the trampolined style one can switch back to SQL and express the function through a recursive CTE.

Figure 3.4 shows an implementation of `fib(n)` in a FP language:

```
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Figure 3.4: fibonacci sequence in FP style.

3.2 Continuation-Passing Style (CPS)

The first step to get closer to an iterative function is to convert the potentially (non-)linear recursive function into a *tail recursive* function. Tail recursive functions are linear recursive functions where the recursive call is the last execution of the function [6].
A positive side effect for tail recursive functions is that the call stack does not need

[2] In this thesis each SQL code line originates from PostgreSQL 13.0.

to be used since no recursive call has to wait for the results of subsequent recursive calls. This prevents the maximum stack depth from being exceeded quickly [7].

The CPS-transformed version of `fib` (see Figure 3.5) does not return [8]. Instead it takes an extra argument, the so called *continuation* (here: `k`). The continuation is a function that expects one argument and is applied at the very end to the calculated result [8]. So the idea is that the continuation carries the plan for the rest of the computation at any time [9].

```
fib :: Int -> (Int -> Int) -> Int
fib 1 k = k 1
fib 2 k = k 1
fib n k = fib (n-1) (\f1 -> fib (n-2)
                        (\f2 -> k (f1 + f2)))
```

Figure 3.5: CPS-transformed function `fib`.

After applying the CPS-Transformation all function calls are tail calls [10]. So the second call `fib (n-2)` has to be moved inside the continuation. Here, the evaluation order is explicit i.e. `fib (n-2)` could also be evaluated before `fib (n-1)`. Therefore one would instead have to move `fib (n-1)` inside the continuation. But the former seems to be a plausible evaluation order.
After evaluating `f1 + f2` the continuation has to be applied to this sum. Otherwise the remaining computation would be dismissed.

The problem is that the CPS-Transformation leaves a higher-order function since `fib` now expects a function `k` as parameter. SQL can only represent first-order functions. But fortunately there is a method to eliminate higher-order functions, namely *defunctionalization*.

3.3 Defunctionalization

The basis of defunctionalization is that a program only consists of finitely many function abstractions [11]. These function abstractions are the continuations. One can assign an unique identifier to each of them and instead of transferring function abstractions as arguments one can use those identifiers. That way defunctionalization eliminates higher-order functions by turning them into equivalent first-order functions [11].

3.3.1 Lambda lifting

Since a function abstraction can contain free variables one step has to take place before the actual defunctionalization. The function abstractions have to be lambda lifted to identify the free variables and pass them explicitly. That way they later (see 3.3.2) can be stored together with the identifier. The lambda lifting turns local lambda abstractions into global functions [12].
In function `fib` there are two lambda abstractions, namely `(\f1 -> fib (n-2)(\f2 -> k (f1 + f2)))` and `(\f2 -> k (f1 + f2))` [3]. After the lambda lifting (see Figure 3.6) these local lambda abstractions become the global functions `fibNext` and `fibAdd`. `fibNext` computes the next fibonacci number $fib(n-2)$ and `fibAdd` adds the fibonacci numbers $fib(n-1)$ and $fib(n-2)$.

```
fibNext :: Int -> (Int -> Int) -> Int -> Int
fibNext n k f1 = fib (n-2) (fibAdd f1 k)

fibAdd :: Int -> (Int -> Int) -> Int -> Int
fibAdd f1 k f2 = k (f1 + f2)

fib :: Int -> (Int -> Int) -> Int
fib 1 k = k 1
fib 2 k = k 1
fib n k = fib (n-1) (fibNext n k)
```

Figure 3.6: Lambda lifted version of the fibonacci sequence.

3.3.2 Defunctionalization

The functions are still higher order. To change this, a set of records that identifies the lambda lifted functions (aka continuations) is needed[13]. For this a sum data type `Kont` which provides one constructor for each continuation is created. A constructor (here: `Done`) to represent the empty continuation is also required.
For the fibonacci use case Figure 3.7 shows the sum data `Kont` with the constructors `Done`, `Next` and `Add`.

```
data Kont = Done
          | Next Int Kont
          | Add Int Kont
```

Figure 3.7: Continuation as sum data type `Kont`.

[3] Note that the second lambda abstraction also appears in the first one.

For `Next` and `Add` the free variables that are used in the continuations have to be passed. These are two variables each. One variable is of type `Int`. It is needed for the augend resp. addend (in case of `Next` resp. `Add`). The other variable is needed for the continuation and has type `Kont` [4]. In the program (see Figure 3.8) all calls to lambda lifted functions are replaced by the corresponding constructor of `Kont`.

```
fibNext :: Int -> Kont -> Int -> Int
fibNext n k f1 = fib (n-2) (Add f1 k)

fibAdd :: Int -> Kont -> Int -> Int
fibAdd f1 k f2 = k (f1 + f2) ↯

fib :: Int -> Kont -> Int
fib 1 k = k 1 ↯
fib 2 k = k 1 ↯
fib n k = fib (n-1) (Next n k)
```

Figure 3.8: Incompletely defunctionalized version of the fibonacci sequence.

Higher-order functions are now eliminated but `fib 1 k`, `fib 2 k` and `fibAdd f1 k f2` do not work anymore (↯). `k` is no longer a function but a data type. Thus a separate function `apply` (see Figure 3.9) takes over the application of `k` [13]. It expects two arguments, one of type `Kont` and the other of the overall result type `Int`. `apply` dispatches over the `Kont` constructors [11]. For `Done` the argument of type `Int` will be returned. If the `Kont` parameter is constructed via `Next` the code is already given with `fib (n-2)(Add f1 k)`. Finally, for`Add`, `apply` is called with the given arguments.

```
data Kont = Done
          | Next Int Kont
          | Add Int Kont

fib :: Int -> Kont -> Int
fib 1 k = apply k 1
fib 2 k = apply k 1
fib n k = fib (n-1) (Next n k)

apply :: Kont -> Int -> Int
apply Done x        = x
apply (Next n k) f1 = fib (n-2) (Add f1 k)
apply (Add f1 k) f2 = apply k (f1 + f2)
```

Figure 3.9: Completely defunctionalized version of the fibonacci sequence.

[4] Notice that this means that `Kont` is defined recursively.

For the fibonacci sequence only one dispatcher function is needed. But this does not always has to be the case since e.g. not all function abstractions have to have the same return type. It is therefore not convenient to only use one dispatcher function [11]. In the program code one then has to think of which dispatcher function to use at a specific location.

As with the CPS-Transformation, a new problem arises after defunctionalizing the program. The continuation data type `Kont` is defined recursively because the constructors `Next` and `Add` expect an argument for the free variable `Kont`. This must be changed since a recursive CTE cannot express that.

3.4 User-defined Stack

As presented by Bartosz Milewski [14], to eliminate the recursion in data type `Kont` it can be redefined as a list (see Figure 3.10). The elements of this list are tuples that contain the free variables (except variable `Kont`) together with a reference id (see type `Reference`). This reference id (e.g. an integer) can be treated as the new function identifier similar to the explanation in 3.3.2.

```
type Reference = Int
type Kont = [(Int, Reference)]
```

Figure 3.10: Continuation `Kont` represented as list.

`Kont` does not have to be included in the list elements since the list contains all continuations and is passed at any time. This list imitates a *(user-defined) stack* — `push`, `pop`, `top` and `isEmpty` are the only operations needed.

```
fib :: Int -> Kont -> Int
fib 1 k = apply k 1
fib 2 k = apply k 1
fib n k = fib (n-1) ((n, 1):k)

apply :: Kont -> Int -> Int
apply [] x            = x
apply ((n, 1):k) f1   = fib (n-2) ((f1, 2):k)
apply ((f1, 2): k) f2 = apply k (f1 + f2)
```

Figure 3.11: fibonacci sequence using a user-defined stack.

In function `fib` (see Figure 3.11) either elements get **pushed** to the continuation or the dispatcher function `apply` is called. If `apply` is invoked with an **empty** continuation list, it returns the overall result. Otherwise it consumes the **top** element of the continuation and in case of reference id `1`, the next fibonacci number is **pushed** to the stack. In case of reference id `2` the sum is calculated and the remaining continuation is forwarded.

3.5 Trampolined Style

At the moment, the functions `fib` and `apply` are mutually tail recursive. The corresponding call graph (see Figure 3.12) is too complex for the single-cycle iteration scheme of a recursive CTE [5].

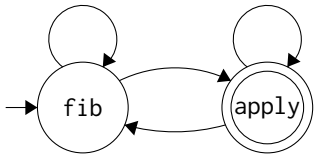

Figure 3.12: Call graph of defunctionalized fibonacci sequence.

Therefore, the following steps still need to be taken to enable a representation by a recursive CTE:

1. Combine the work of the transformed function (here: `fib`) and the `apply` function(s) into one function `tramp`.
2. Remove the recursion of `tramp` by transforming it into the trampolined style.

3.5.1 Inling

This section shows how to combine several functions into one that performs the job of all functions.
Let us recall the fibonacci use case. Currently, there are two functions `fib` and `apply`. To result in only one function `tramp` a new data type `Label` (see Figure 3.13) is introduced. This helps to distinguish between the functions `fib` and `apply`.

```
data Label = Fib | Apply deriving (Eq)
```

Figure 3.13: Data type `Label` to distinguish between functions.

[5] The same problem occured in section 2.2.2

For those who are not familiar with the haskell syntax: `deriving (Eq)` implements a possibility to check the equality of two labels. This property enables to pattern matched on `Label`.

`tramp` gets four parameters:

- `Label` parameter to distinguish between the functions `fib` and `apply`.
- `Int` parameter that `fib` expects.
- `Int` parameter that `apply` expects.
- `Kont` parameter that both functions expect.

Instead of calling `fib` or `apply`, now `tramp` is called with a label (indicating the function intented to use) together with the arguments. In case of receiving label `Fib` the `apply` parameter `a` can be ignored and vice versa for a label `Apply` the `Fib` parameter `f` is ignored. This is further demonstrated through the use of `undefined` in Figure 3.14.

```
tramp :: Label -> Int -> Int -> Kont -> Int
tramp Fib   1 a   k            = tramp Apply undefined 1 k
tramp Fib   2 a   k            = tramp Apply undefined 1 k
tramp Fib   n a   k            = tramp Fib (n-1) undefined ((n, 1):k)
tramp Apply f x   []           = x
tramp Apply f f1 ((n, 1):k)    = tramp Fib (n-2) undefined ((f1, 2):k)
tramp Apply f f2 ((f1, 2): k) = tramp Apply undefined (f1 + f2) k
```

Figure 3.14: Inlining of `fib` and `apply`.

Since the `fib` and `apply` parameters have the same data type (`Int`) and both parameters aren't used simultaneously, the number of parameters can be reduced (see Figure 3.15).

```
tramp :: Label -> Int -> Kont -> Int
tramp Fib 1 k            = tramp Apply 1 k
tramp Fib 2 k            = tramp Apply 1 k
tramp Fib n k            = tramp Fib (n-1) ((n, 1):k)
tramp Apply x []         = x
tramp Apply f1 ((n, 1):k) = tramp Fib (n-2) ((f1, 2):k)
tramp Apply f2 ((f1, 2): k) = tramp Apply (f1 + f2) k
```

Figure 3.15: Reduced number of parameters after Inlining.

3.5.2 Trampolined Style

When transforming `tramp` into the trampolined style the recursion has to be removed and instead the program has to be executed in a single loop by producing one computation at one step [15]. The loop continues until a result of `tramp` indicates to be the overall result. Hence another label constructor (here: `Finish`) is needed.

```
data Label = Fib | Apply | Finish deriving (Eq)
```

Figure 3.16: Data type `Label` with additional constructor `Finish`.

Figure 3.17 presents the function `tramp` in the trampolined style. Every self-invocation of `tramp` is removed by a tuple containing the new label and the arguments that would be passed in the self-invocation. In every case where `tramp` once just returned the overall result, this result is wrapped into the tuple format together with the end label `Finish`.

```
tramp :: Label -> Int -> Kont -> (Label, Int, Kont)
tramp Fib 1 k            = (Apply, 1, k)
tramp Fib 2 k            = (Apply, 1, k)
tramp Fib n k            = (Fib, n-1, (n, 1):k)
tramp Apply n []         = (Finish, n, [])
tramp Apply f1 ((n, 1):k) = (Fib, n-2, (f1, 2):k)
tramp Apply f2 ((f1, 2): k) = (Apply, f1+f2, k)
```

Figure 3.17: Function `tramp` in the trampolined style.

Compared to SQL a separate helper function `driver` (see Figure 3.18) is needed in Haskell. `driver` has the task to repeatedly execute `tramp` until it returns a tuple containing the end label. Unpacking this tuple yields the overall result.

```
driver :: Int -> Int
driver n = res
    where (_,res,_) = until (\(label,_,_) -> label == Finish)
                            (uncurry3 tramp)
                            (Fib, n, [])

-- Uncurry a Triplet
uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
uncurry3 f (a, b, c) = f a b c
```

Figure 3.18: Function `driver` imitates SQL's `WITH RECURSIVE` construct.

Every invocation of `tramp` only creates the subsequent tuple containing the next label, argument and continuation. If `driver` wants to know the n-th fibonacci number it starts to invoke `tramp` with the label `Fib`, argument n and the empty continuation `[]` (see Line 5 of Figure 3.18). `uncurry3` extracts these arguments out of the tuple and calls `tramp` with these parameters. This procedure continues until `tramp` outputs a tuple with the label `Finish`. Then `driver` returns the overall result.

Figure 3.19 illustrates the interplay of `driver` and `tramp`. This exactly matches the single-loop iteration scheme of a recursive CTE. Thus the program can be represented through a `WITH RECURSIVE` construct.

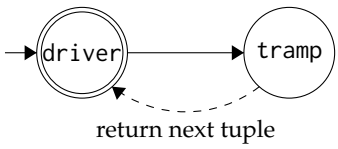

Figure 3.19: Single-loop iteration scheme.

3.6 WITH RECURSIVE

Now it is time to change the programming language back to SQL. This trampolined style version can easily be expressed in SQL using a recursive CTE (see Figure 3.20). Here, an extra helper function `driver` is not needed because (as discussed in 2.1) the recursive CTE is already implemented as a loop that continues until a produced working table is empty. `WITH RECURSIVE` can be seen as the `driver` that generates rows until the overall result is calculated and a `LATERAL`-Join can perform the job of the function `tramp`.
The recursive CTE produces only one row per run since the `SELECT-WHERE` blocks in `tramp` are mutually exclusive. The last produced row `d` can be grabbed per `LATERAL`-Join and according to the properties of `d` the next row is produced. In the post-processing a filter enables to only return the column `res` where the label is `'Finish'`.

```sql
CREATE TYPE kont AS (num numeric, ref int);

CREATE FUNCTION fib(n numeric) RETURNS numeric AS
$$
WITH RECURSIVE driver(label, res, k) AS (
  SELECT 'Fib', n, ARRAY[] :: kont[]
    UNION ALL
  SELECT _.*
  FROM  driver AS d, LATERAL
         (SELECT 'Apply', 1, d.k
         WHERE d.label = 'Fib' AND (d.res = 1 OR d.res = 2)
           UNION ALL
         SELECT 'Fib', d.res-1, (d.res,1) :: kont || d.k
         WHERE d.label = 'Fib' AND d.res > 2
           UNION ALL
         SELECT 'Finish', d.res, d.k
         WHERE d.label = 'Apply' AND CARDINALITY(d.k) = 0
           UNION ALL
         SELECT 'Fib', d.k[1].num - 2, (d.res, 2) :: kont || d.k[2:]
         WHERE d.label = 'Apply' AND k[1].ref = 1
           UNION ALL
         SELECT 'Apply', d.res + d.k[1].num, d.k[2:]
         WHERE d.label = 'Apply' AND d.k[1].ref = 2
         ) AS tramp
) SELECT d.res FROM driver AS d WHERE d.label = 'Finish';
$$ LANGUAGE SQL STABLE STRICT;
```

Figure 3.20: Fibonacci sequence represented through a recursive CTE.

4 OPTIMIZATIONS

After performing the transformation steps seen in chapter 3, the obtained program is expressible through a recursive CTE. This was the whole goal of the transformations. But when turning the right screws, one can likely gain a performance boost by avoiding to perform duplicate work (4.1) and representing information in a more efficient way (4.2). Other optimizations (4.3) mainly result in a reduction of the memory capacity.

4.1 Memoization

So far, the implementation does not benefit if same calculations occur multiple times. Two identical calculations have to perform the whole calculation and can't just access the result of a previously performed identical calculation. This needs to be changed. Two additional parameters are needed:

- `dict` contains entries `(args,res)` that map a concrete combination of arguments `args` to its calculated result `res`.
- `current` works as a stack where the top element is the combination of arguments `args` for which the result `res` is currently calculated.

Using those parameters, one can lookup `args` in the dictionary and finds out whether `res` has already been stored or the calculation actually has to be performed. In the latter case `args` is pushed to `current`. After `res` has been calculated the top element of `current` is popped and stored in the dictionary together with `res`.

4.1.1 JSONB-Dictionary

A way to represent the dictionary is to use a *JSONB* array. It can be created by casting a string of the form '{"k_1" : v_1, $\cdots$,"k_n" : v_n}' to the SQL data type `jsonb`. Given a key k_i one can then retrieve the corresponding value v_i. Therefore one has to use the following syntax: `<jsonb> ->>` k_i [1]

The corresponding implementation of the fibonacci sequence (see Figure 4.1) works like this:

- Grab current row `d`
- If `d.label = 'Fib'`:
  - Do a lookup in the dictionary
    - `IF` a corresponding entry exists: Skip the calculation and apply the retrieved value directly
    - `ELSE`: Push the current combination of arguments to `current`
  - The JSONB dictionary is comma-separated
    - `IF` a base case is reached (`d.res` $\in \{1, 2\}$) `AND` the dictionary is empty: Add the new key-value-pair **without** a leading comma
    - `ELSE`: Add the new key-value-pair **with** a leading comma
- If `d.label = 'Apply'`
  - After calculating a value insert it in the dictionary together with the top element of `current` as the key
  - Pop this key from `current`
  - A distinction between an empty and a non-empty dictionary is not necessary because the base cases must have already inserted an entry to the dictionary

[1] "->>" yields a text value that can be casted to the desired atomic type.

```
CREATE TYPE kont AS (num numeric, ref int);

CREATE FUNCTION fib(n numeric) RETURNS numeric AS
$$
WITH RECURSIVE driver(label, res, k, current, dict) AS (
  SELECT 'Fib', n, ARRAY[] :: kont[], ARRAY[] :: numeric[], '{}' :: text
    UNION ALL
  SELECT _.*
  FROM driver AS d, LATERAL
         (SELECT _.*
         FROM
           (SELECT (d.dict :: jsonb ->> (d.res :: text)) :: numeric) AS lookup(val),
           LATERAL (SELECT 'Apply', lookup.val, d.k, d.current, d.dict
                  WHERE d.label = 'Fib' AND lookup.val IS NOT NULL
                    UNION ALL
                  SELECT 'Apply', calc.res, d.k, d.current,
                        (left((d.dict :: text), -1) || '"' || d.res || '":' || calc.res || '}') :: text
                  FROM (SELECT 1 :: numeric) AS calc(res)
                  WHERE d.label = 'Fib' AND lookup.val IS NULL AND d.dict = '{}' AND (d.res = 1 OR d.res = 2)
                    UNION ALL
                  SELECT 'Apply', calc.res, d.k, d.current,
                        (left((d.dict :: text), -1) || ', "' || d.res || '":' || calc.res || '}') :: text
                  FROM (SELECT 1 :: numeric) AS calc(res)
                  WHERE d.label = 'Fib' AND lookup.val IS NULL AND d.dict <> '{}' AND (d.res = 1 OR d.res = 2)
                    UNION ALL
                  SELECT 'Fib', d.res-1, (d.res, 1) :: kont || d.k, d.res || d.current, d.dict
                  WHERE d.label = 'Fib' AND lookup.val IS NULL AND d.res > 2
                  ) AS _
           UNION ALL
         SELECT 'Finish', d.res, d.k, d.current, d.dict
         WHERE d.label = 'Apply' AND CARDINALITY(d.k) = 0
           UNION ALL
         SELECT 'Fib', d.k[1].num - 2, (d.res, 2) :: kont || d.k[2:], d.current, d.dict
         WHERE d.label = 'Apply' AND d.k[1].ref = 1
           UNION ALL
         SELECT 'Apply', calc.res, d.k[2:], d.current[2:],
               (left((d.dict :: text), -1) || ', "' || d.current[1] || '":' || calc.res || '}') :: text
         FROM (SELECT d.res + d.k[1].num) AS calc(res)
         WHERE d.label = 'Apply' AND d.k[1].ref = 2
         ) AS tramp
) SELECT d.res FROM driver AS d WHERE d.label = 'Finish';
$$ LANGUAGE SQL STABLE STRICT;
```

Figure 4.1: Fibonacci sequence using a JSONB dictionary.

4.1.2 Hash-Table-Dictionary

The JSONB representation of the dictionary entails several disadvantages:

- Need for an extra column to transfer the dictionary during the loop of the recursive CTE
- Transfering the dictionary over two consecutive loops causes the dictionary to be recreated from scratch (even if no changes are made to it)
- Need to distinguish between empty and non-empty dictionary
- Detour by casting a string to JSONB and finally to the desired atomic type
- Special treatment for inserting a `NULL` value since JSONB does not accept a value to be just `NULL`

These circumstances call for an alternative representation of the dictionary. Using the programming language *C*, one can implement an extension to create hash tables which can then be used as dictionaries.

The implemented C-Extension comes with the following functions:

- Legend:
  - `<table_id>`: integer that identifies hash table
  - `<#keys>`: number of key columns in hash table
  - `<type`$_i$`>`: value of arbitrary type `t` (e.g. `NULL :: t`) indicating type of key column i
  - `<key`$_i$`>`: value of key column i
  - `<value`$_i$`>`: value to be inserted at the i-th value column
- Functions:
  - `prepareHT(<table_id>, <#keys>, <type`$_1$`>, ···, <type`$_n$`>)`
    - creates a hash table with the given properties
  - `lookupHT(<table_id>, <key`$_1$`>, ···, <key`$_n$`>)`
    - returns the value for a given key in hash table `<table_id>`
  - `insertToHT(<table_id>, <key`$_1$`>, ···, <key`$_n$`>, <value`$_1$`>, ···, <value`$_n$`>)`
    - inserts a key-value pair in hash table `<table_id>`
  - `removeFromHT(<table_id>, <key`$_1$`>, ···, <key`$_n$`>)`
    - removes a key-value pair in hash table `<table_id>`
  - `lengthHT(<table_id>)`
    - returns the number of entries in hash table `<table_id>`

For the fibonacci sequence a hash table with table_id 1 is used as dictionary. The number of key columns is 1 and its type is `numeric`. The value column is also of type `numeric`.

```
SELECT prepareHT(1, 1, NULL :: numeric, NULL :: numeric);
```

Figure 4.2: Hash table dictionary for the fibonacci sequence.

Hash table 1 is now active during the whole SQL-Session i.e. the hash table dictionary does not have to be explicitly passed around. Thus the column `dict` can be removed from the recursive CTE.

The hash table implementation of the fibonacci sequence (see Figure 4.3) uses an additional `LEFT OUTER JOIN` for the lookup. This guarantees that the recursive CTE can produce the next row even if the lookup does not find a corresponding entry in the hash table dictionary. Without the `LEFT OUTER JOIN`, `lookupHT` would return `void` and the `LATERAL JOIN` would therefore produce no row.

The dictionary insertions take place in the `FROM` clauses. This is an appropriate place (besides the `SELECT` clause) compared to the `WHERE` clauses. Here, the `WHERE` clauses will be evaluated before the `FROM` clauses since they are independent of each other [2].
Another point against dictionary insertions in the `WHERE` clauses is that the predicates in the `WHERE` clause can change their evaluation order and an insertion should only take place when **all** predicates evaluate to true. For example, the insertions could not be moved to the end of the `WHERE` clauses because SQL does not guarantee that the insertions only take place when all predicates evaluate to true.

[2] In the SQL evaluation plan this is indicated by a `Result` node.

```
CREATE FUNCTION fib(n numeric) RETURNS numeric AS
$$
WITH RECURSIVE driver(label, res, k, current) AS (
  SELECT 'Fib', n, ARRAY[] :: kont[], ARRAY[] :: numeric[]
    UNION ALL
  SELECT _.*
  FROM driver AS d, LATERAL
      (SELECT _.*
      FROM
       (SELECT NULL) AS ___ LEFT OUTER JOIN lookupHT(1, d.res) AS lookup(num
           numeric, val numeric) ON TRUE,
       LATERAL (SELECT 'Apply', lookup.val, d.k, d.current
              WHERE d.label = 'Fib' AND lookup.num IS NOT NULL
                UNION ALL
              SELECT 'Apply', calc.res, d.k, d.current
              FROM (SELECT 1 :: numeric) AS calc(res),
                  LATERAL (SELECT insertToHT(1, d.res, calc.res)) AS _
              WHERE d.label = 'Fib' AND (d.res = 1 OR d.res = 2) AND
                  lookup.num IS NULL
                UNION ALL
              SELECT 'Fib', d.res-1, (d.res, 1) :: kont || d.k, d.res ||
                  d.current
              WHERE d.label = 'Fib' AND d.res > 2 AND lookup.num IS NULL
              ) AS _
       UNION ALL
      SELECT 'Finish', d.res, d.k, d.current
      WHERE d.label = 'Apply' AND CARDINALITY(d.k) = 0
       UNION ALL
      SELECT 'Fib', d.k[1].num - 2, (d.res, 2) :: kont || d.k[2:], d.current
      WHERE d.label = 'Apply' AND d.k[1].ref = 1
       UNION ALL
      SELECT 'Apply', calc.res, d.k[2:], d.current[2:]
      FROM (SELECT d.k[1].num + d.res) AS calc(res),
          LATERAL (SELECT insertToHT(1, d.current[1], calc.res)) AS _
      WHERE d.label = 'Apply' AND d.k[1].ref = 2
      ) AS tramp
) SELECT d.res FROM driver AS d WHERE d.label = 'Finish';
$$ LANGUAGE SQL STABLE STRICT;
```

Figure 4.3: Fibonacci sequence using a hash table dictionary.

4.2 Hash-Table-Arrays

Some problems that the JSONB dictionary has also concern arrays in SQL:

- Need for extra columns to transfer the arrays `k` and `current` during the loop of the recursive CTE
- Transfering an array over two consecutive loops causes the array to be recreated from scratch (even if no changes are made to it)

In terms of runtime, especially long arrays can suffer from these circumstances. A hash table representation of the arrays can address this issue because hash tables are persistent throughout a SQL session. Hence they do not have to be explicitly transfered.

For the fibonacci sequence, three hash tables are necessary to represent the dictionary and the arrays `k` resp. `current`:

```
SELECT prepareHT(1, 1, NULL :: numeric, NULL :: numeric);
SELECT prepareHT(2, 1, NULL :: int, NULL :: numeric, NULL :: int);
SELECT prepareHT(3, 1, NULL :: int, NULL :: numeric);
```

Figure 4.4: Hash table dictionary and hash table arrays for the fibonacci sequence.

The hash table dictionary again uses table id `1`. table id `2` belongs to the continuation `k` and hash table with table id `3` replaces the array `current`. An element of an array is defined by its unique position and its values. Hence, in the hash tables with table id `2` and `3`, the position is used as the key. Since the elements of the arrays `k` and `current` were tuples the components inside a tuple are split into individual value columns.

Next, `fib` shows the implementation of the fibonacci sequence using a hash table dictionary and hash table arrays:

```sql
CREATE FUNCTION fib(n numeric) RETURNS numeric AS
$$
WITH RECURSIVE driver(label, res, cmds, no_cache) AS (
  SELECT 'Fib', n, NULL :: text, 1 :: int
    UNION ALL
  SELECT _.*
  FROM driver AS d, LATERAL
      (SELECT _.*
      FROM
       (SELECT NULL) AS ___ LEFT OUTER JOIN lookupHT(1, d.res) AS lookup(num numeric, val numeric) ON TRUE,
       LATERAL (SELECT 'Apply', lookup.val, NULL, d.no_cache
              WHERE d.label = 'Fib' AND lookup.num IS NOT NULL
                UNION ALL
              SELECT 'Apply', calc.res, insertToHT(1, d.res, calc.res) :: text, d.no_cache
              FROM (SELECT 1 :: numeric) AS calc(res)
              WHERE d.label = 'Fib' AND (d.res = 1 OR d.res = 2) AND lookup.num IS NULL
                 UNION ALL
              SELECT 'Fib', d.res - 1,
                     insertToHT(2, lengthHT(d.no_cache + 2 - d.no_cache), d.res, 1) :: text ||
                     insertToHT(3, lengthHT(d.no_cache + 3 - d.no_cache), d.res) :: text, d.no_cache
              WHERE d.label = 'Fib' AND d.res > 2 AND lookup.num IS NULL
              ) AS _
       UNION ALL
      SELECT 'Finish', d.res, NULL, d.no_cache
      WHERE d.label = 'Apply' AND lengthHT(d.no_cache + 2 - d.no_cache) = 0
       UNION ALL
      SELECT __.*
      FROM
       lookupHT(2, lengthHT(d.no_cache + 2 - d.no_cache) - 1) AS k(idx int, num numeric, ref int),
       LATERAL (SELECT 'Fib', k.num - 2, insertToHT(2, k.idx, d.res, 2) :: text, d.no_cache
              WHERE k.ref = 1
                 UNION ALL
              SELECT 'Apply', d.res + k.num, insertToHT(1, c.num, d.res + k.num) :: text || removeFromHT(2, k.idx) :: text || removeFromHT(3, c.idx) :: text, d.no_cache
              FROM lookupHT(3, lengthHT(d.no_cache + 3 - d.no_cache) - 1) AS c(idx int, num numeric)
              WHERE k.ref = 2
              ) AS __
      WHERE d.label = 'Apply'
      ) AS tramp
) SELECT d.res FROM driver AS d WHERE d.label = 'Finish';
$$ LANGUAGE SQL STABLE STRICT;
```

Although the columns `k` and `current` aren't needed anymore, the number of columns in the recursive CTE does not decrease. The reasons for that are the additional columns `cmds` and `no_cache`. Column `cmds` is needed because the hash table operations `insertToHT` and `removeFromHT` have to be moved into the `SELECT` clause. An explanation of this follows later. Column `no_cache` contains a dummy value that is used to prevent SQL from returning a cached result and instead reevaluate the desired function call.

All the necessary changes can be explained by looking at the lower part of the implementation (Lines 33-35).
The last entry `c` in `current` has to be looked up first, then `c` is stored as key in the dictionary together with the result as value. Only after that `c` can be removed from `current`. So it is important to first do the lookup and then remove this entry. To keep this order `insertToHT` and `removeFromHT` are moved to the `SELECT` clause and the lookup takes place in the `FROM` clause. This plan works because the `SELECT` clause is evaluated after the other clauses.
The hash table operations used in the `SELECT` clause are declared to return `VOID`. So these functions have to be casted to an atomic type. The `text` type is appropriate for this. If multiple hash table operations have to be performed, they can easily be concatenated using `'||'`.

To explain the use of column `no_cache`, take a look at line 29. If you would call `lengthHT(2)` you would receive a length x and PostgreSQL would then cache this result. Then, according to line 30 you would insert an entry to this hash table. Now the unexpected: When calling `lengthHT(2)` again you would not receive $x + 1$ but x. This is due to the fact that PostgreSQL cached the value of `lengthHT(2)` at the first call.
This is not what we want. So the dummy value (here: `1`) from the column `no_cache` helps out. When calling `lengthHT(d.no_cache + 2 - d.no_cache)` PostgreSQL reevaluates the function for every call and does not return a cached value.

4.3 WITH ITERATIVE

For the UDF in trampolined style and the optimized UDFs (4.1, 4.2) each loop of the recursive CTE produces exactly one row. The row produced in the last loop of the recursive CTE automatically contains the label `'Finish'` and therefore entails the final result of the UDF.
So it is not necessary to collect all produced rows in the result table of the recursive CTE only to iterate over this potentially huge result table in the postprocessing and search for the row with the label `'Finish'`.

The space requirement for the result table and the time to maintain it and then traverse it in the postprocessing can be reduced by a small change of the recursive CTEs. Replacing `WITH RECURSIVE` by `WITH ITERATIVE` sets exactly there. `WITH ITERATIVE` does not maintain a result table but passes the last produced non-empty working table to the postprocessing.
The advantage of `WITH ITERATIVE` compared to `WITH RECURSIVE` is on the one hand a reduction of the runtime but above all a large space saving. However, `WITH ITERATIVE` is not available by default inside PostgreSQL v13 but must be added itself.

Since the runtime reductions are marginal, `WITH ITERATIVE` is not considered further in the runtime experiments 5.

5 EXPERIMENTS AND DISCUSSION

This chapter analyzes the effects of the SQL-to-SQL compilation described in chapter 3 and takes the promising optimizations (4.1, 4.2) into consideration.
Two experiments provide details about the performance of UDFs **without** (5.2) and **with** (5.3) memoization.

5.1 Setup

The following experiments are performed using PostgreSQL v13.0 on a 64-bit Linux x86 host. It uses 2 AMD EPYC™ 7402 CPUs with a base clock rate of 2.8GHz and maximum power clock rate up to 3.35GHz. The DDR4 RAM has a size of 512 GB where 128MB are used for caching (`shared_buffers = 128MB`). Notable changes to the server configuration file `postgresql.conf` are the increases of `work_mem = 512MB` (to prevent as far as possible that temporary results are materialized on disk) and `max_stack_depth = 7680kB` (to allow more recursive calls that can be pushed to the call stack) [16].

On the basis of 13 different use cases (provided by [2]), the naive functional-style UDF is compared to the corresponding UDF in trampolined style and the corresponding UDFs that implement the optimizations proposed in chapter 4. Additionally the compiled UDF discussed in the paper *Functional-Style SQL UDFs With a Capital 'F'* [2] (mentioned in 2.2.1) is thrown into the pot.

The use cases (see Table 5.1) represent a diverse portfolio of different problems. Among them are mathematical functions (`fib`, `fac`, `mandel`), graph algorithms (`comps`, `floyd`), string processings (`fsm`, `lcs`), hierarchical functions (`paths`, `sizes`), interpretations of expressions and programs (`eval`, `vm`), a classification method (`dtw`) and a 2D object recognition (`march`).

UDF	Description	Recursion
`mandel`	compute Mandelbrot set	tail
`paths`	reconstruct path names in a file system	tail
`sizes`	aggregate file sizes in a directory hierarchy	tail
`vm`	run a program on a simple virtual machine	tail
`fac`	$fac(0) = 1, n \in \mathbb{N} : fac(n) = n \cdot (n-1) \cdot \ldots \cdot 1$	linear
`fsm`	parse molecule names using a state machine	linear
`march`	trace border of 2D object (Marching Squares)	linear
`comps`	test for connected components in a DAG	2-fold
`eval`	evaluate arithmetic expressions	2-fold
`fib`	compute n-th fibonacci number	2-fold
`lcs`	find longest common subsequence of strings	2-fold
`dtw`	measure distance between two time series (Dynamic Time Warping)	3-fold
`floyd`	find length of shortest path (Floyd-Warshall)	3-fold

Table 5.1: 13 functional-style UDFs with a short description and their type of recursion.

Legend for the experiments:

- **naive**
  - functional-style UDF
- **F**
  - compiled UDF proposed by *Functional-Style SQL UDFs With a Capital 'F'*
- **tramp**
  - UDF in **trampolined style**
  - Representation of continuation parameter `k`: **Array**
- **memo(JSONB + Arrays)**
  - UDF in **trampolined style**
  - Representation of the dictionary: **JSONB**
  - Representation of parameters `k` and `current`: **Arrays**
- **memo(HT + Arrays)**
  - UDF in **trampolined style**
  - Representation of the dictionary: **Hash table**
  - Representation of parameters `k` and `current`: **Arrays**
- **memo(HT + HTs)**
  - UDF in **trampolined style**
  - Representation of the dictionary: **Hash table**
  - Representation of parameters `k` and `current`: **Hash tables**

5.2 Runtime comparison w/o Memoization

For each use case, this experiment provides a graph indicating the runtimes of the UDFs listed in the previously seen legend. The timings are averaged over five runs, with worst and best time disregarded.
For the mathematical use cases `dtw`, `fac` and `fib` the x-axis denotes the input argument. All other use cases display the number of invocations on the x-axis.
On the y-axis, all figures show the averaged runtimes. The y-axis is logarithmically scaled to better visualize the runtime differences. Moreover, it allows exponential growth to be quickly detected since it results in a straight line for a logarithmically scaled y-axis and a linearly scaled x-axis.

All UDFs besides the naive functional-style UDF and the UDF in trampolined style use a form of memoization. But here the memoization is (explicitly) discarded after one invocation of the UDF. The next invocation then starts again without memoized results. However, if an invocation itself would have formerly produced recursive calls, the computations associated with these calls can profit from memoized results. But this is an advantage of the memoization variants and therefore intentional. These variants update the dictionary immediately after each computed (intermediate) result.
This runtime comparison helps to better compare the different UDFs without taking advantage of memoized results from previous invocations.

Additionally, there is a table for each use case that informs about the speedup for every UDF. Say UDF u has a speedup of s i.e. in the time that the functional-style UDF computed its result, u could compute the result s times.

5.2.1 Analysis

The recursion type of a functional-style UDF has a deep impact on this experiment. This section partitions the use cases into their types of recursion, plots their runtimes and gives an overall analysis for each recursion type.

Since the runtime behaviors of the use cases are partly similar within a recursion type, not all graphs of all use cases are shown in this chapter. However, the missing graphs can be examined in the appendix 1.

5.2.1.1 Tail Recursion

If a functional-style UDF is already tail recursive, some transformation steps (CPS, defunctionalization, user-defined stack) can be omitted. The goal of these omitted transformation steps is just to produce a tail recursive function that does not use higher-order functions and recursive data types. If a tail recursive function can be represented by a UDF these steps aren't necessary. Tail recursive UDFs can directly be transformed into the trampolined style.

Due to the missing CPS-Transformation, the UDFs in trampolined style do not have a continuation parameter `k` and after invoking the UDF all formerly produced tail calls have the same result. Therefore the variants memo(HT + Arrays) and memo(HT + HTs) are implemented slightly different for tail recursive functional-style UDFs.

The result of each produced tail call is known only **after** the last tail call calculated the final result. Hence, during the evaluation all intermediate argument combinations have to be collected. memo(HT + Arrays) uses an array and memo(HT + HTs) uses a hash table for the collection. After calculating the final result the collected argument combinations are stored together with the final result in a hash table dictionary.

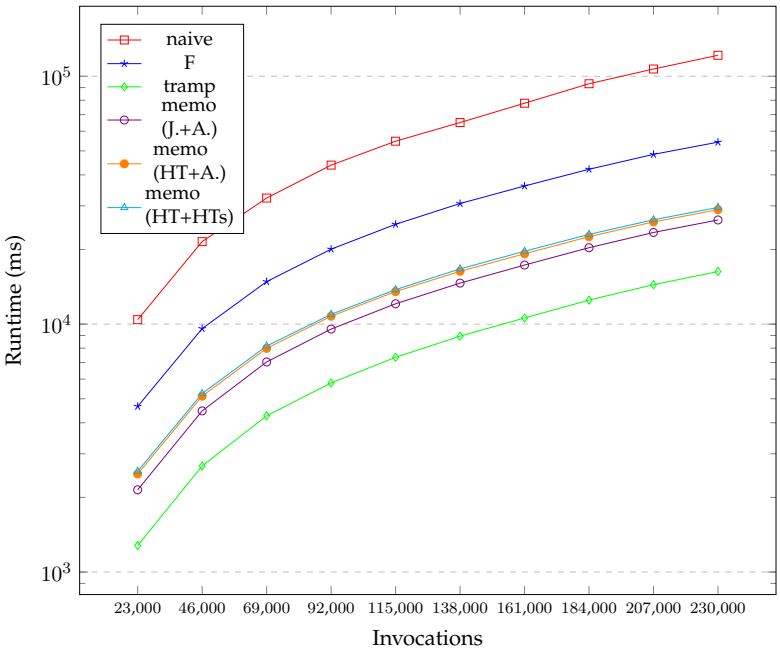

Figure 5.1: `paths`: runtime comparison w/o memoization.

UDF	Speedup
F	2.19
tramp	7.46
memo (JSONB + Arrays)	4.57
memo (HT + Arrays)	4.11
memo (HT + HTs)	4.02

Figure 5.2: `paths`: speedup.

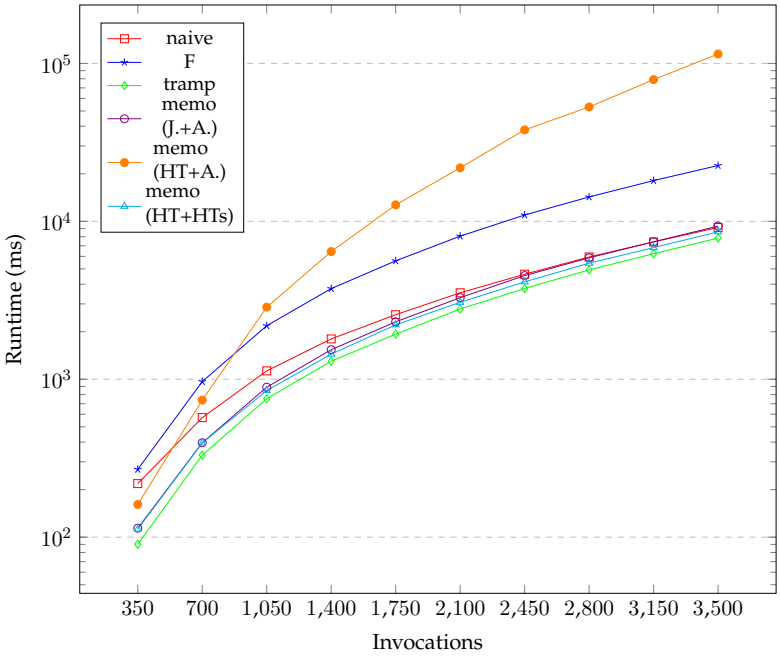

Figure 5.3: `sizes`: runtime comparison w/o memoization.

UDF	Speedup
F	0.43
tramp	1.23
memo (JSONB + Arrays)	1.03
memo (HT + Arrays)	0.11
memo (HT + HTs)	1.12

Figure 5.4: `sizes`: speedup.

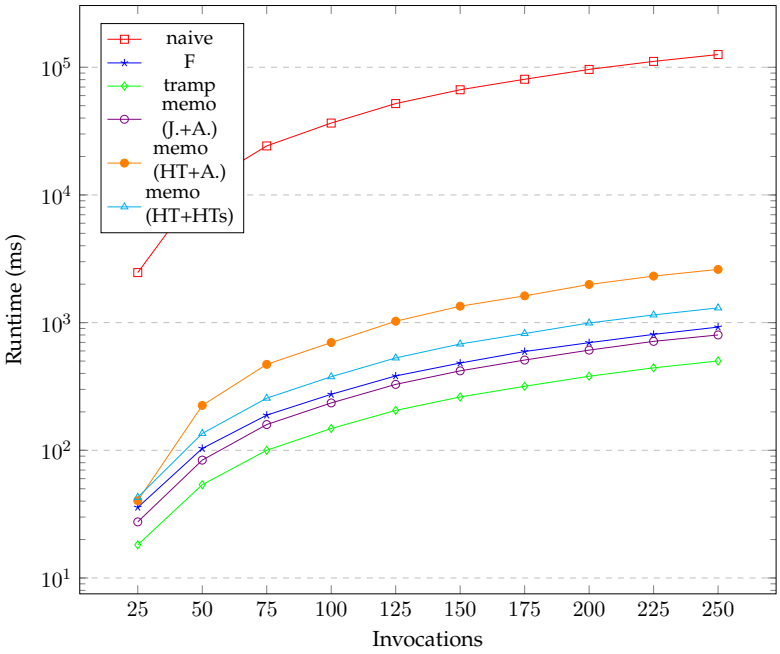

Figure 5.5: `vm`: runtime comparison w/o memoization.

UDF	Speedup
F	135.33
tramp	250.17
memo (JSONB + Arrays)	156.36
memo (HT + Arrays)	49.22
memo (HT + HTs)	96.60

Figure 5.6: `vm`: speedup.

The discussed advantages of a recursive CTE over the naive functional-style UDF are reflected in the clearly more performant UDF tramp. tramp provides an (enormous) speedup in each of the use cases (from 1.23 for `sizes` up to 250.17 for `vm`). The runtime in `sizes` is mainly influenced by a complex array aggregation. This is the reason why the transformations have a comparatively low impact on the runtime.
Here, the optimizations from chapter 4 (implemented in memo(JSONB + Arrays), memo(HT + Arrays) and memo(HT + HTs)) offer no benefit compared to tramp. That's because a call to a tail recursive function would never produce a recursive call with an already used combination of arguments.
In all use cases memo(HT + Arrays) is slower than memo(JSONB + Arrays). This is due to the fact that memo(JSONB + Arrays) does not fill its dictionary with all intermediate argument combinations after calculating the overall result. For memo(JSONB + Arrays) this wouldn't make sense because a jsonb dictionary is not persistent and therefore never used after calculating the final result.
In `sizes`, memo(HT + HTs) is faster than memo(JSONB + Arrays) because this use case produces comparatively few recursive calls. Thus fewer argument combinations have to be collected and so the advantages of a hash table over jsonb prevail.

5.2.1.2 Linear Recursion

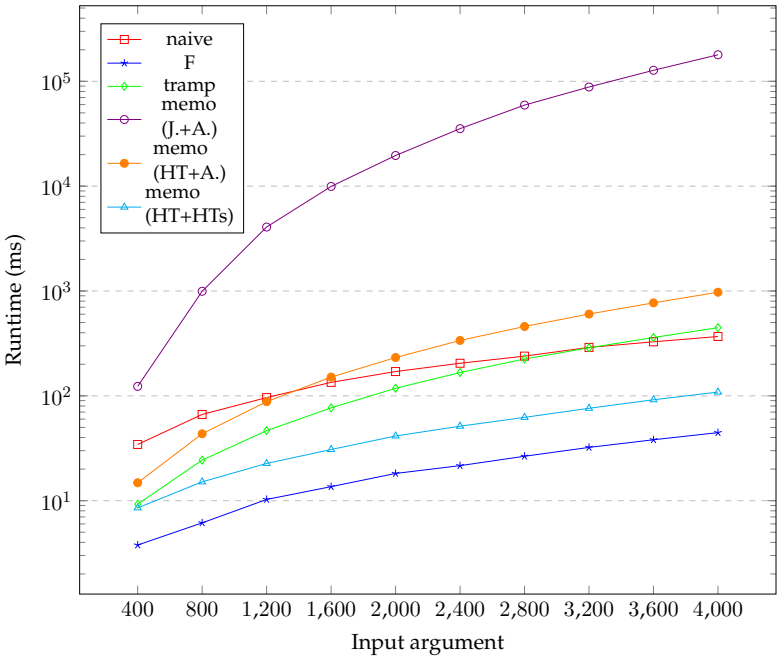

Figure 5.7: `fac`: runtime comparison w/o memoization.

UDF	Speedup
F	8.98
tramp	1.10
memo (JSONB + Arrays)	0.00
memo (HT + Arrays)	0.53
memo (HT + HTs)	3.80

Figure 5.8: `fac`: speedup.

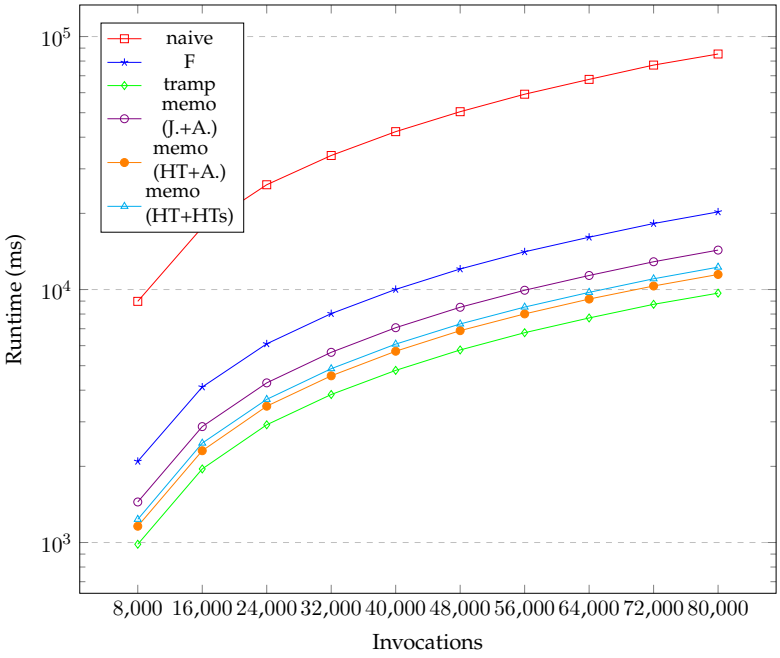

Figure 5.9: `fsm`: runtime comparison w/o memoization.

UDF	Speedup
F	4.21
tramp	8.81
memo (JSONB + Arrays)	5.98
memo (HT + Arrays)	7.43
memo (HT + HTs)	6.97

Figure 5.10: `fsm`: speedup.

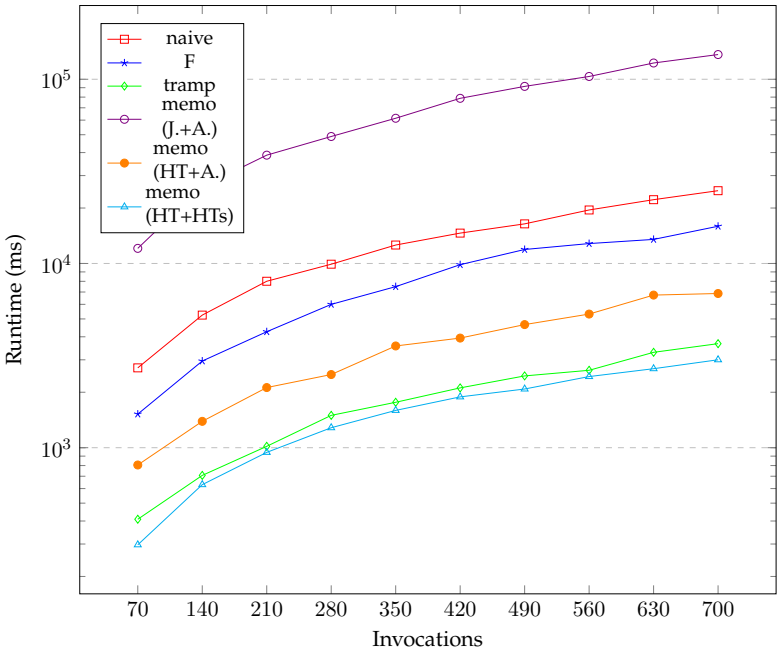

Figure 5.11: `march` runtime comparison

UDF	Speedup
F	1.58
tramp	6.96
memo (JSONB + Arrays)	0.19
memo (HT + Arrays)	3.59
memo (HT + HTs)	8.08

Figure 5.12: `march` speedup

Regarding tramp and F the transformation of the naive functional-style UDF into a recursive CTE pays off. In use case `fac` the variant of F uses a template to exploit the linear recursion resulting in a massive speedup of about 9.
As with tail recursion, linear recursion does not produce recursive calls with an already used combination of arguments. So the additional work that memo(HT + Arrays) and memo(JSONB + Arrays) perform is wasted effort since a dictionary lookup will never return a corresponding entry.
Inserting entries into a jsonb dictionary is much more expensive than inserting into a hash table dictionary. This leads to the bad performance of memo(JSONB + Arrays). However, in use case `fsm` memo(JSONB + Arrays) does not perform bad. This is due to the fact that `fsm` is very easily represented as a tail recursive functional-style UDF and based on this functional-style UDF the variants tramp, memo(JSONB + Arrays), memo(HT + Arrays) and memo(HT + HTs) have been implemented. Therefore, these variants have similar properties as the UDFs discussed in section 5.2.1.1.
memo(HT + HTs) performes very well. That's because the parameters `k` and `current` can become very large for linear recursive use cases. Using hash tables instead of arrays for these parameters cause the high speedup.

5.2.1.3 2-fold Recursion

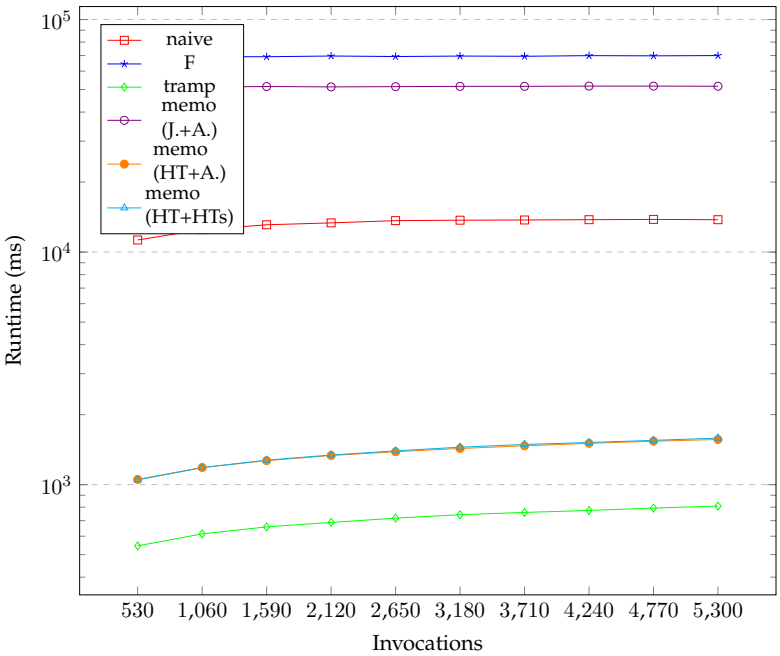

Figure 5.13: `eval`: runtime comparison w/o memoization.

UDF	Speedup
F	0.19
tramp	18.69
memo (JSONB + Arrays)	0.26
memo (HT + Arrays)	9.67
memo (HT + HTs)	9.59

Figure 5.14: `eval`: speedup.

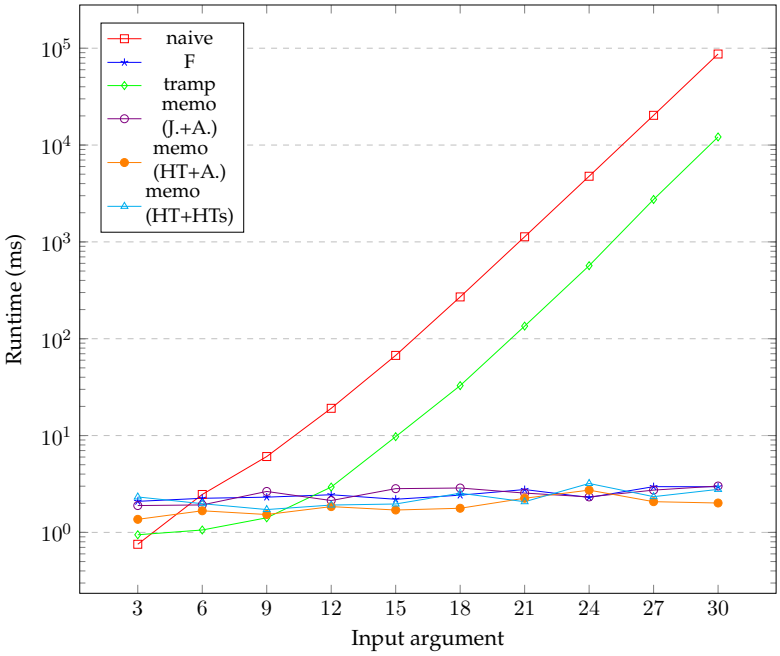

Figure 5.15: `fib`: runtime comparison w/o memoization.

UDF	Speedup
F	4598.30
tramp	7.28
memo (JSONB + Arrays)	4561.63
memo (HT + Arrays)	5993.77
memo (HT + HTs)	4976.21

Figure 5.16: `fib`: speedup.

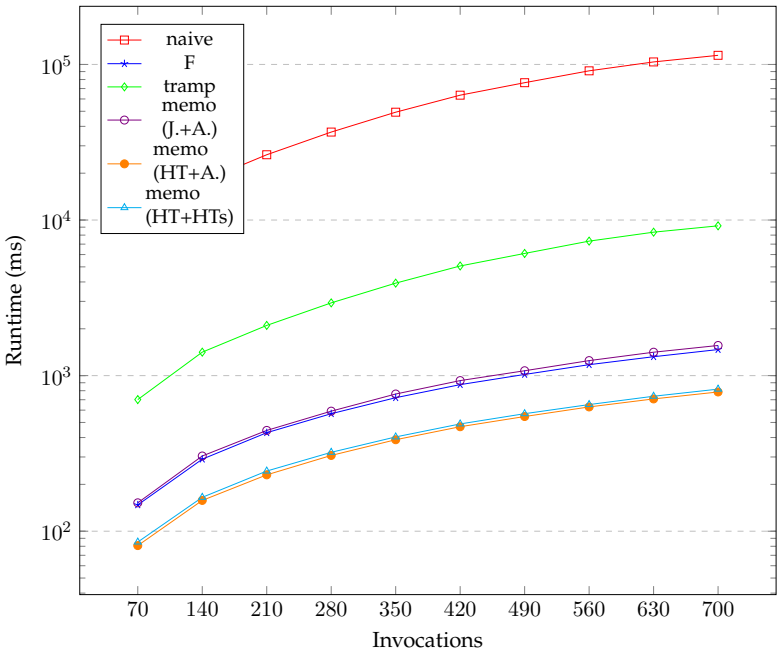

Figure 5.17: `lcs`: runtime comparison w/o memoization.

UDF	Speedup
F	73.29
tramp	12.48
memo (JSONB + Arrays)	69.29
memo (HT + Arrays)	136.67
memo (HT + HTs)	131.07

Figure 5.18: `lcs`: speedup.

The advantages of recursive CTEs over naive are noticeable since tramp entails a decent speedup for each use case.
Now that the functional-style UDFs use 2-fold recursion the memoization finally is worth the effort. There is now the chance that a recursive call has already done the work of another. Especially for the use cases `fib` and `lcs` many dictionary lookups can return already calculated results. memo(JSONB + Arrays), memo(HT + Arrays) and memo(HT + HTs) all have even higher speedups than tramp for these two use cases. `fib` is an almost perfect use case for memoization resulting in a speedup of up to ≈ 6000. The variant F of `fib` uses the reference counting optimization mentioned in 2.2.1. That's why the high speedup of ≈ 4600 occurs here.
The use case `eval` doesn't use already calculated results with the same frequency as `fib` and `lcs`. Therefore, the speedup of memo(JSONB + Arrays), memo(HT + Arrays) and memo(HT + HTs) compared to tramp turns out sobering. memo(JSONB + Arrays) is even slower than naive because despite the expensive maintenance of the jsonb dictionary, corresponding entries are rarely found here.
All use cases do not produce many entries for the parameters `k` and `current`. Hence, there is no performance boost from memo(HT + HTs) over memo(HT + Arrays).

5.2.1.4 3-fold Recursion

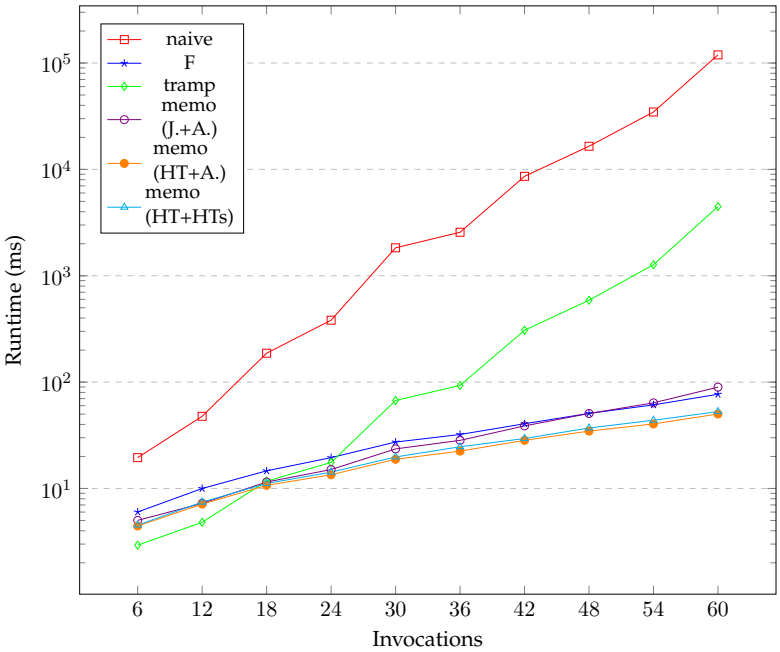

Figure 5.19: `dtw`: runtime comparison w/o memoization.

UDF	Speedup
F	542.89
tramp	26.94
memo (JSONB + Arrays)	551.65
memo (HT + Arrays)	798.43
memo (HT + HTs)	751.29

Figure 5.20: `dtw`: speedup.

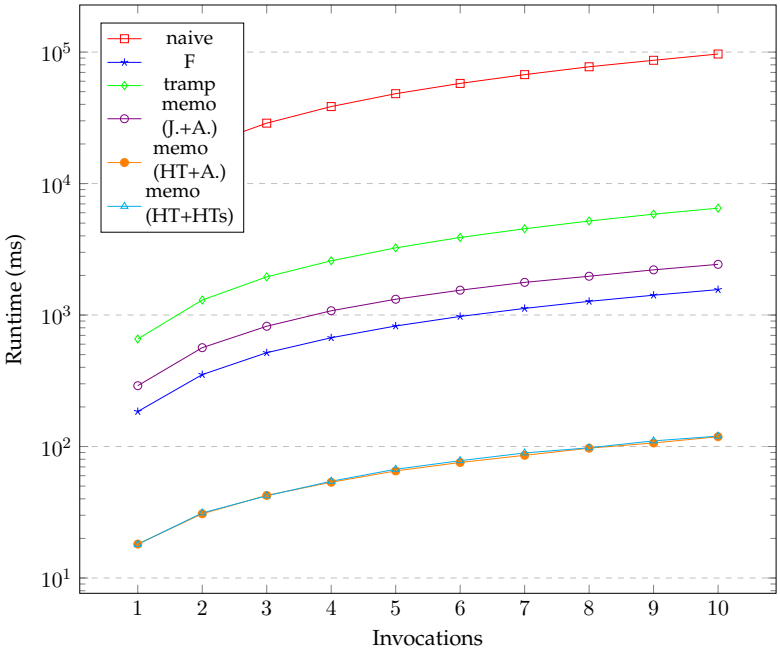

Figure 5.21: `floyd` runtime comparison

UDF	Speedup
F	59.58
tramp	14.85
memo (JSONB + Arrays)	37.87
memo (HT + Arrays)	764.14
memo (HT + HTs)	748.13

Figure 5.22: `floyd` speedup

All transformations cause a high performance boost over the naive functional-style UDF. The positive effect of the memoization in memo(JSONB + Arrays), memo(HT + Arrays) and memo(HT + HTs) compared to tramp is very strongly strengthened by the 3-fold recursion since in both use cases the already calculated results are often needed several times.
The high speedup of F in the use case `dtw` is due to the fact that (as for use case `fib`) the reference counting optimization is used in this variant.

5.3 Runtime comparison w/ Memoization

This experiment focuses on the effects of memoization.
For each analyzed use case two graphs are generated. One graph displays the runtimes and the other graph displays the memoization entries over a sequence of random calls without discarding the memoization table.
The linearly scaled x-axis denotes the measuring point in the sequence. To avoid a zigzag shape of the runtimes a measuring point consists of a set of calls, a so-called *batch*. The batch size n_{batch} is different for each use case and is selected in such a way that the memoization effect can be recognized well. The plotted runtime is the

sum of the runtimes from the n_{batch} calls of the corresponding measuring point.
For the same reasons as in the first experiment (5.2 Runtime comparison w/o Memoization), the runtimes resp. memoization entries on the y-axis are scaled logarithmically.

In the runtime graph, the variants F, tramp, memo(JSONB + Arrays), memo(HT + Arrays) and memo(HT + HTs) are plotted to analyze the impact of memoization in a direct comparison. The naive functional-style UDF is left out of this experiment because no memoization effect is expected for this variant anyway.
The memoization entry graph only depicts variants F, memo(HT + Arrays) and memo(HT + HTs). These are the only variants that use a persistent memoization table. tramp on the other hand does not use memoization at all and for memo(JSONB + Arrays) the scope of the used dictionary is limited to a single call of the UDF. Therefore, it is unsuitable to compare the entries of the jsonb dictionary with the persistent alternatives of F, memo(HT + Arrays) and memo(HT + HTs).

5.3.1 Analysis

This section describes the usual course of runtimes and memoization entries depending on the number of measuring points. Specific specialties of individual use cases are also explained.
Memoization has the same effect for most use cases. Therefore, this section does not discuss each of the 13 use cases individually. The graphs of the omitted use cases can be found in the appendix 2.

For tramp and memo(JSONB + Arrays) no change in runtime is expected over several measurement points. This is obvious for tramp because no dictionary is maintained. memo(JSONB + Arrays) on the other hand, maintains a dictionary. But since the dictionary is used as a column in the recursive CTE, it only has this limited scope. Hence, each call of the UDF has to construct its own dictionary and no memoization effect can take place between two calls.
These variants are nevertheless included in this experiment in order to be able to compare them to variants that use a persistent memoization table.
Variants that maintain a persistent memoization table are F, memo(HT + Arrays) and memo(HT + HTs). In general, a positive memoization effect in terms of a runtime reduction is expected due to the reuse of already computed results. However, this effect varies in intensity for each use case.

5.3.1.1 Usual Effect

The usual course of the graphs can be explained very well on the basis of the use cases `vm` and `floyd`.

The largest runtime reduction occurs early on and settles at a stable level as the measuring points increase. Convergence is achieved faster by memo(HT + Arrays) and memo(HT + HTs) than by F since the former update their dictionary immediately after a new result is computed. In variant F the dictionary is only updated after the final result has been calculated.

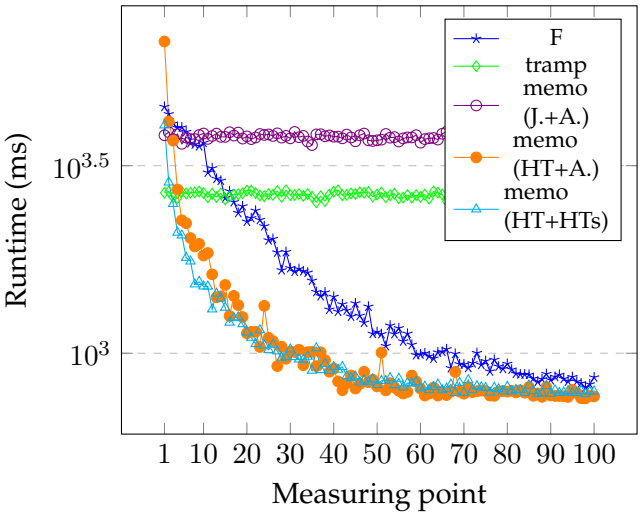

Figure 5.23: `vm`: runtime comparison w/ memoization.

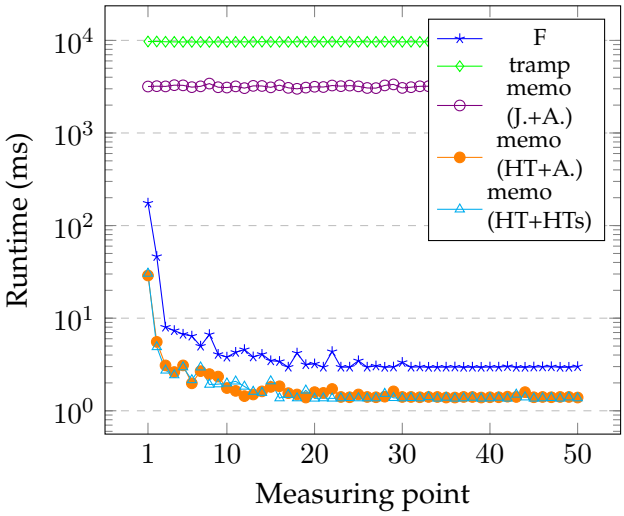

Figure 5.24: `floyd`: runtime comparison w/ memoization.

For tail recursive use cases (see `vm`) and linear recursive use cases in general, tramp and memo(JSONB + Arrays) can keep up with F, memo(HT + Arrays) and memo(HT + HTs) in terms of runtimes when the number of measuring points is still small. In this case the lookups simply provide too few suitable entries to recognize an effect. For most 2-fold or 3-fold recursive use cases (see `floyd`), the variants that use a persistent memoization table are already more performant for small measuring points.

Regardless of the recursion type, with an increasing number of measuring points the use of a persistent memoization table prevails.

The memoization entry graphs behave similarly. Here however, not the largest reduction but the largest increase in memoization entries occurs for small measuring points. This value also converges after several measuring points.

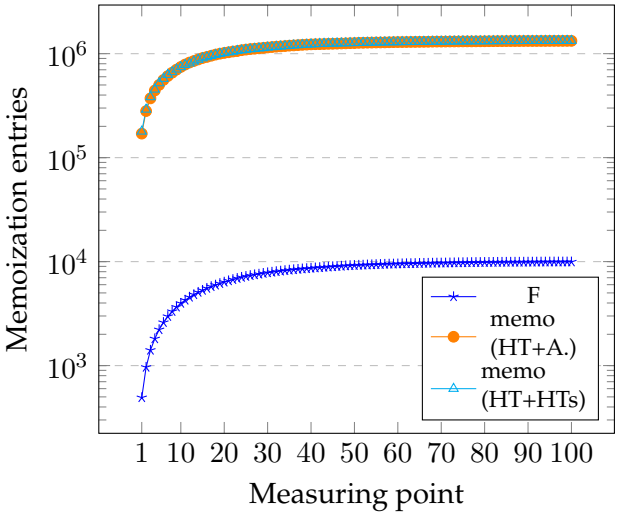

Figure 5.25: `vm`: comparison of memoization entries.

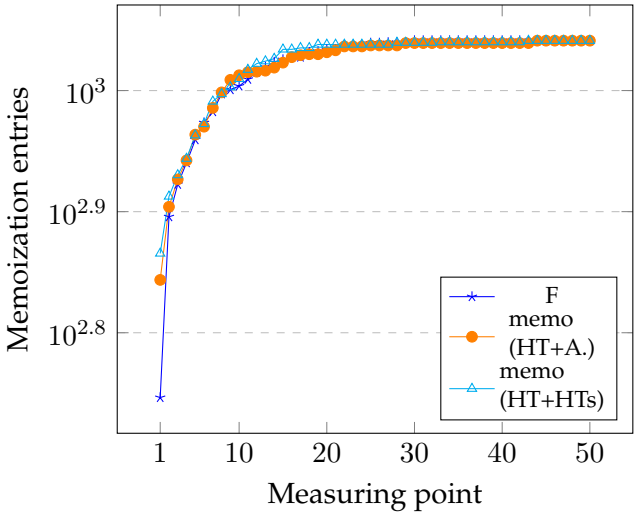

Figure 5.26: `floyd`: comparison of memoization entries.

For the use cases whose functional-style UDFs are tail recursive (see `vm`) F has less memoization entries than memo(HT + Arrays) and memo(HT + HTs). The reason for this is that F does not store the arguments of all tail calls but only the arguments of the last tail call. With memo(HT + Arrays) and memo(HT + HTs) all arguments are entered into the dictionary as soon as the final result is calculated.

The correlation between the increase in the number of memoization entries and the reduction in runtime is striking. This makes sense, because with a higher number of memoization entries there is also a higher chance that a lookup has a corresponding result ready.
Since the maximum number of possible argument combinations in the experiments is limited, the number of memoization entries is also limited. Hence, the graphs converge with an increasing number of measuring points. At the beginning there are still many unknown argument combinations and at the end many of them have occurred before and are therefore already contained in the dictionary. Looking back at the use cases whose functional-style UDFs are tail recursive, this further explains why the higher numbers of memoization entries for memo(HT + Arrays) and memo(HT + HTs) lead to an earlier convergence of the runtime than for F.

5.3.1.2 Special Observations

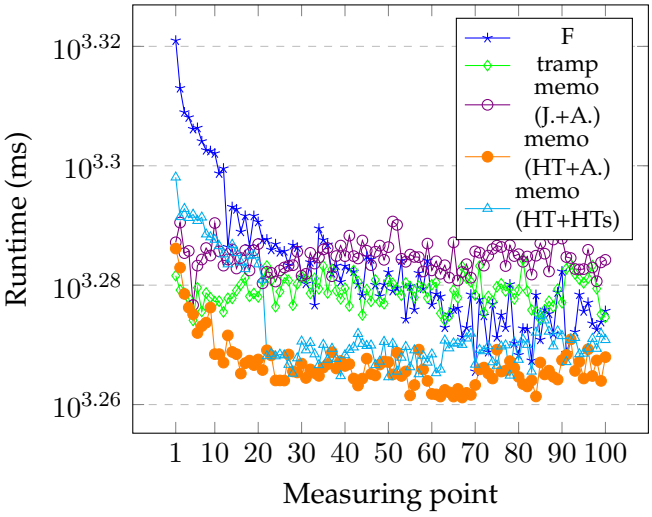

Figure 5.27: `paths`: runtime comparison w/ memoization.

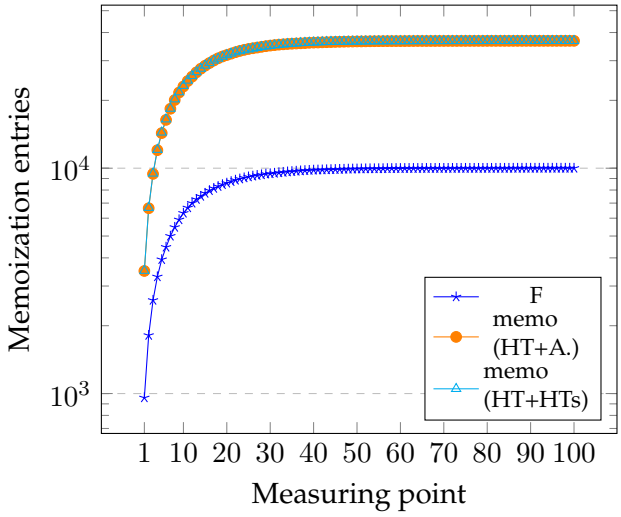

Figure 5.28: `paths`: comparison of memoization entries.

Use case `paths` obtains a runtime reduction through memoization, however the effect is minimal. That's because `paths` only benefits from memoization if the UDF is repeatedly called with the exact same arguments.

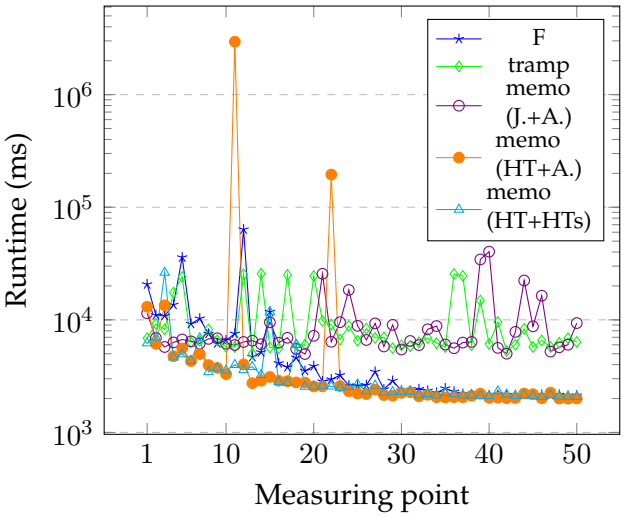

Figure 5.29: `sizes`: runtime comparison w/ memoization.

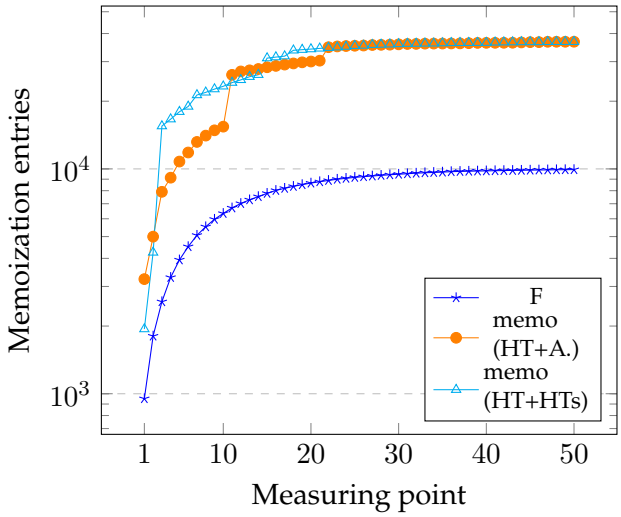

Figure 5.30: `sizes`: comparison of memoization entries.

For use case `sizes` the runtime reduction through memoization is somewhat

stronger because `sizes` also benefits from intermediate results of other calls. However, the effect is still rather small since `sizes` (as well as `paths`) produces relatively few recursive calls.

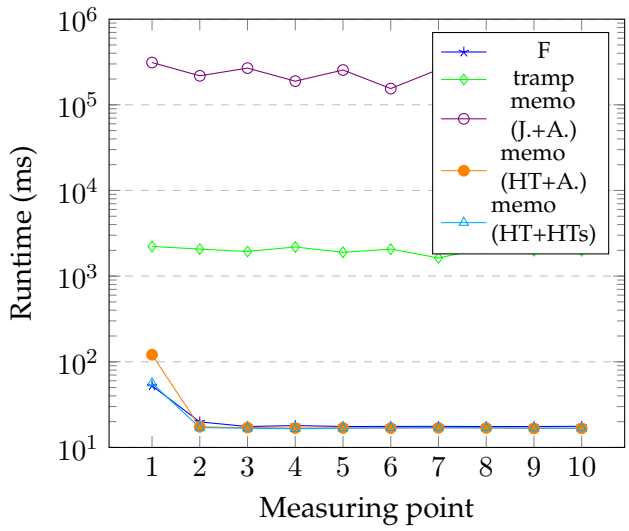

Figure 5.31: `fac`: runtime comparison w/ memoization.

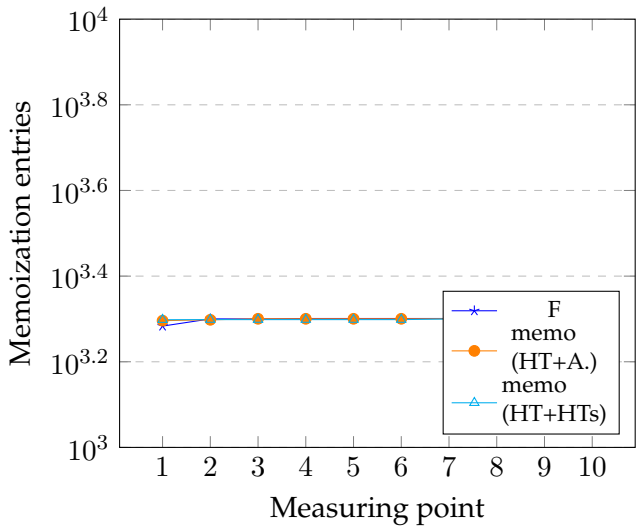

Figure 5.32: `fac`: comparison of memoization entries.

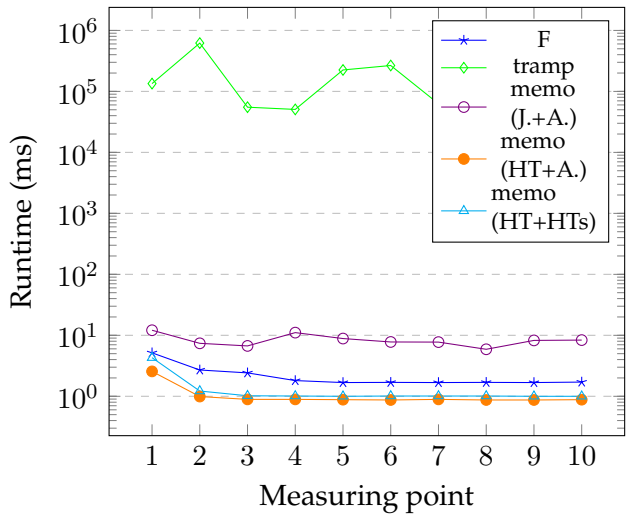

Figure 5.33: `fib`: runtime comparison w/ memoization.

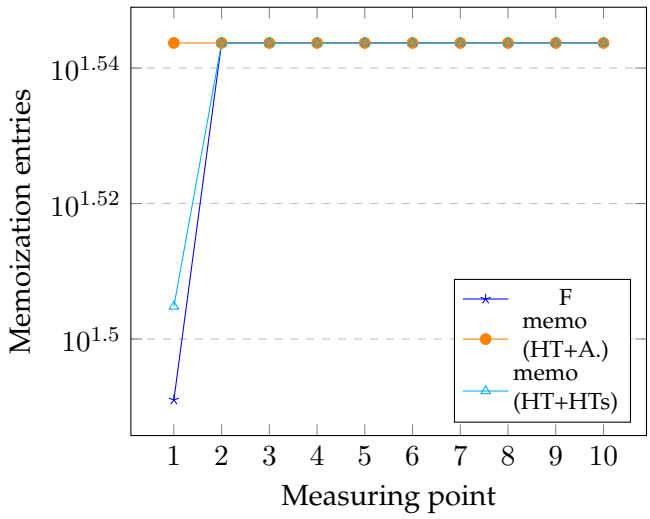

Figure 5.34: `fib`: comparison of memoization entries.

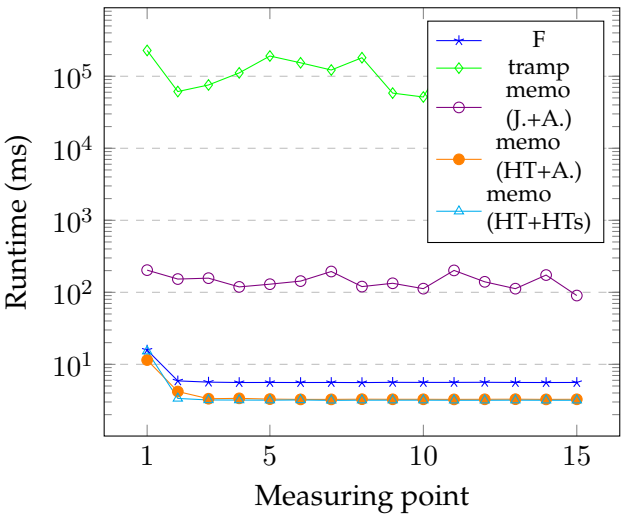

Figure 5.35: `dtw`: runtime comparison w/ memoization.

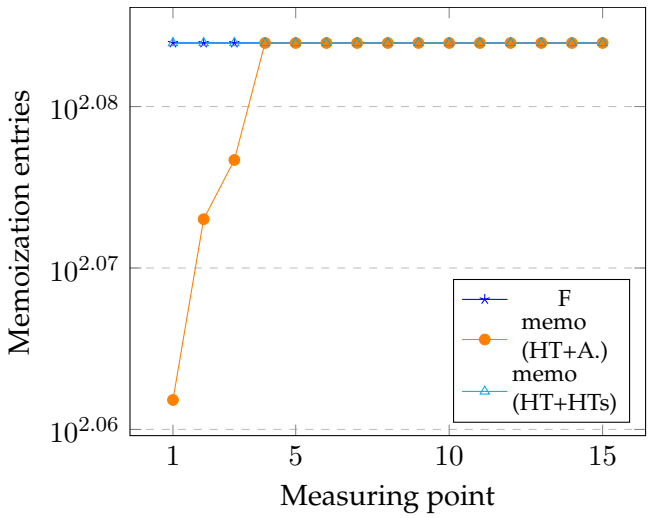

Figure 5.36: `dtw`: comparison of memoization entries.

Due to their definitions, the use cases `fac`, `fib` and `dtw` benefit extremely fast from memoization. This can easily be explained using the definition of use case `fac`:

$$fac(n) = \begin{cases} 1 & n = 0 \\ n \cdot fac(n-1) & \forall n \geq 1, n \in \mathbb{N} \end{cases}$$

Figure 5.37: Definition of the `fac`.

If $fac(n)$ is called, the results for $fac(n-1), fac(n-2), \ldots, fac(0)$ must be calculated. All these results then end up in the dictionary. For subsequent calls, this usually leads to a considerable reduction of work since a large number of computations has already been performed.
The same applies for the use cases `fib` and `dtw`.

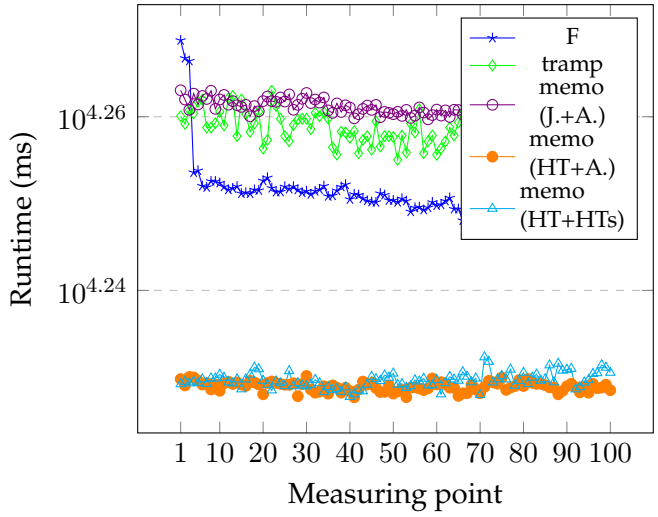

Figure 5.38: `fsm`: runtime comparison w/ memoization.

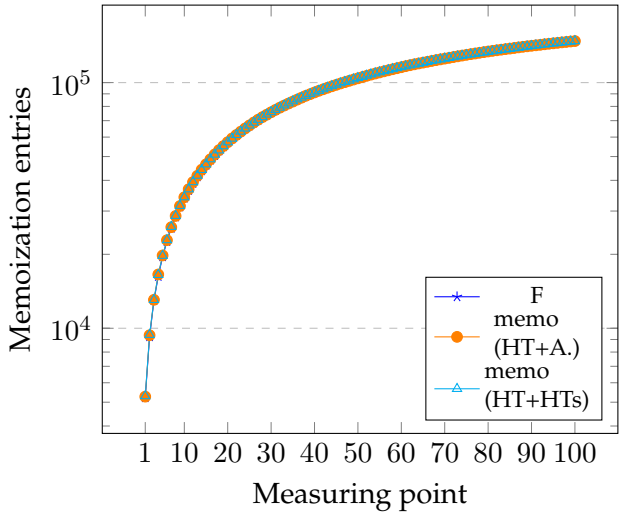

Figure 5.39: `fsm`: comparison of memoization entries.

The use case `fsm` parses chemical compounds. If the formulas of two chemical compounds have the same suffix, memoization can lead to a runtime reduction. In `fsm` the number of recursive calls is measured by the number of characters in the formula of a chemical compound. Therefore, there are very few recursive calls, which in turn makes the possibility of a memoization effect equally small and leads to a convergence of the runtime extremely fast. For memo(HT + Arrays) and memo(HT + HTs) the convergence is reached within the first batch.
Also noticeable in use case `fsm` is the following. As discussed in 5.2.1.2 the memo(HT + Arrays) and memo(HT + HTs) are derived from the tail recursive functional-style UDF of `fsm`. Therefore, one would expect that the number of memoization entries differs from F. This is not the case since the number of recursive calls is extremely low. So the additional entries do not make a noticeable difference.

6 CONCLUSION

6.1 Summary

Using the discussed approach an inefficient functional-style UDF can be transformed into an equivalent and efficient recursive CTE.
The experiments in chapter 5 prove that the SQL-to-SQL compilation leads to a significant runtime reduction both without and with memoization. In part, only memoization can make practical use of highly recursive use cases with high time complexity really usable.

The steps to be performed in this compilation are:

- Required steps:
  1. **Plain Function**
     - turns functional-style UDF into recursive function in FP style
     - ↯ **Problem**: No problem recursion is not expressable with a recursive CTE
  2. **CPS-Transformation**
     - turns (non-)linear recursive functions into tail recursive functions
     - ↯ **Problem**: produces higher-order functions
  3. **Defunctionalization**
     - eliminates higher-order functions
     - ↯ **Problem**: produces recursive data types and mutually tail recursive functions
  4. **User-defined Stack**
     - eliminates recursive data types
     - ↯ **Problem**: function is still (tail) recursive
  5. **Trampolined Style**
     - turns multiple mutually tail recursive functions into one iterative function

        - ✓ No problem to express iterative function with a recursive CTE
- Optional steps:
    - **Memoization**
        - stores and reuses already computed results
        - avoids the performance of duplicate work
    - **Hash-Table-Arrays**
        - create session-persistent representations of arrays
        - avoids array copy operations

The compilation can be applied to UDFs that have scalar (not: tabular) return types. In order to host the approach on a RDBMS it should support recursive CTEs as well as `LATERAL`-Joins and the creation of extensions or at least data structures that provide operations (similar to) `push`, `pop`, `top` and `isEmpty` (stacks, arrays, ...).

Looking back at the problems of RDBMSs mentioned in 1.1, they have now been solved:

- **PostgreSQL**: No replanning of the recursive UDF body leads to a significant performance boost.
- **Microsoft SQL Server** and **Oracle**: The maximum UDF recursion depth is no problem since no call stack is built up.
- **MySQL** and **HyPer**: The prohibition of functional-style UDFs is circumvented by representing them as recursive CTEs.
- **SQLite3**: The non-support of UDFs is solved by the fact that recursive CTEs can also occur as a standalone query.

6.2 Future Work

The use cases discussed in chapter 5 were all compiled manually. So a next step would be to build the corresponding compiler.

But there is also place for possible improvements regarding the implementation. We do not need the full potential that hash tables bring along — stacks are fully sufficient. Thus, in a further extension one could implement a way of creating stacks and use them instead of hash tables.

An additional technique that could be investigated is *parallelization*. Especially for n-fold recursion ($n \geq 2$), the parallel processing of calls could reduce runtime.
Recall the defintion of the 2-fold recursion of the fibonacci sequence:

$$fib(n) = \begin{cases} 1 & n \in \{1, 2\} \\ fib(n-1) + fib(n-2) & \forall n > 2, n \in \mathbb{N} \end{cases}$$

The idea is to evaluate $fib(n-1)$ and $fib(n-2)$ in parallel. However, it is important that all parallel calls use the same dictionary to ensure that the probability of performing duplicate work is negligible.

APPENDIX

1 Runtime comparison w/o Memoization

1.1 Tail-Recursion

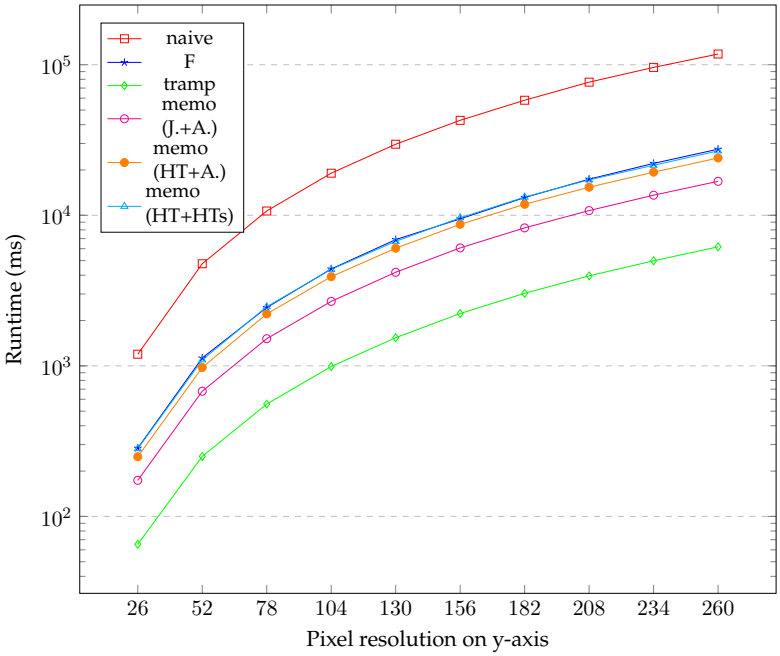

Figure 1: `mandel`: runtime comparison w/o memoization.

UDF	Speedup
F	4.36
tramp	19.20
memo (JSONB + Arrays)	7.06
memo (HT + Arrays)	4.93
memo (HT + HTs)	4.42

Figure 2: `mandel`: speedup.

1.2 2-fold Recursion

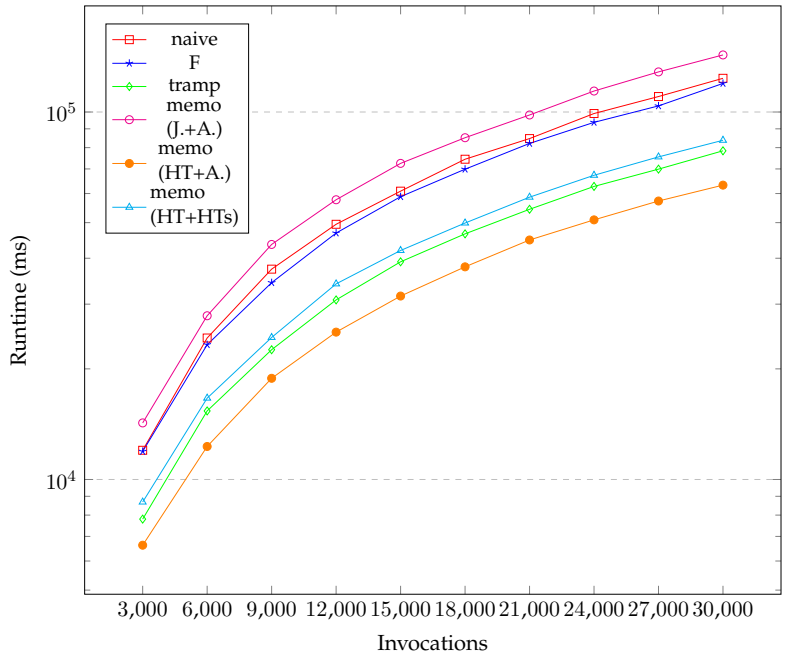

Figure 3: `comps`: runtime comparison w/o memoization.

UDF	Speedup
F	1.05
tramp	1.58
memo (JSONB + Arrays)	0.86
memo (HT + Arrays)	1.94
memo (HT + HTs)	1.47

Figure 4: `comps`: speedup.

2 Runtime comparison w/ Memoization

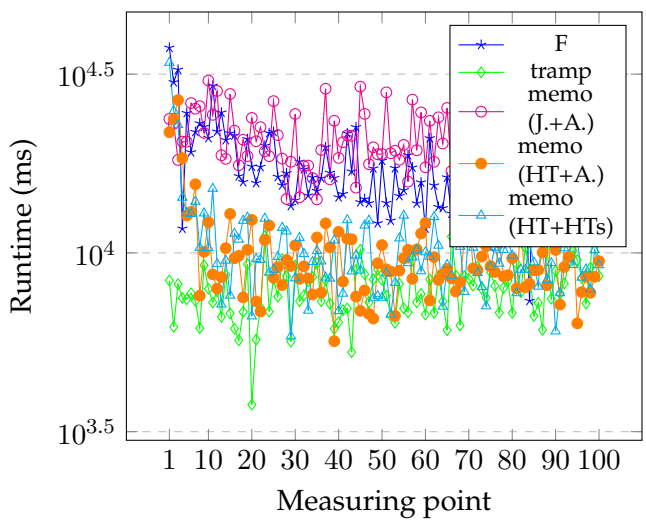

Figure 5: `mandel`: runtime comparison w/ memoization.

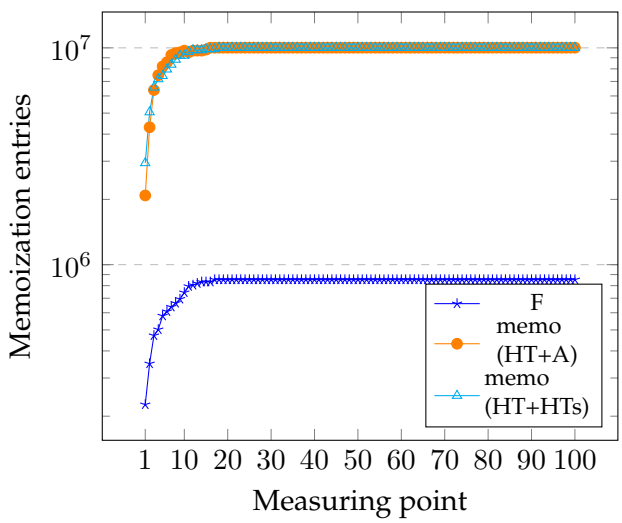

Figure 6: `mandel`: comparison of memoization entries.

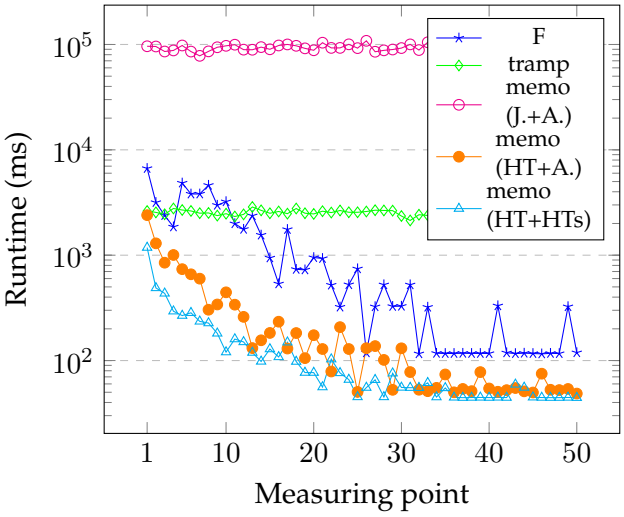

Figure 7: `march`: runtime comparison w/ memoization.

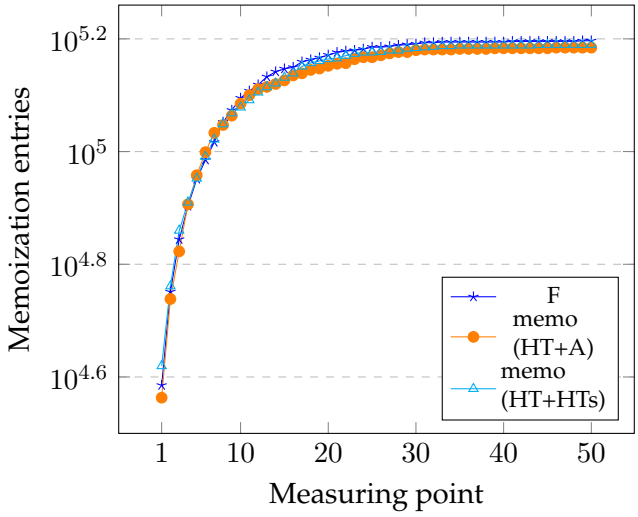

Figure 8: `march`: comparison of memoization entries.

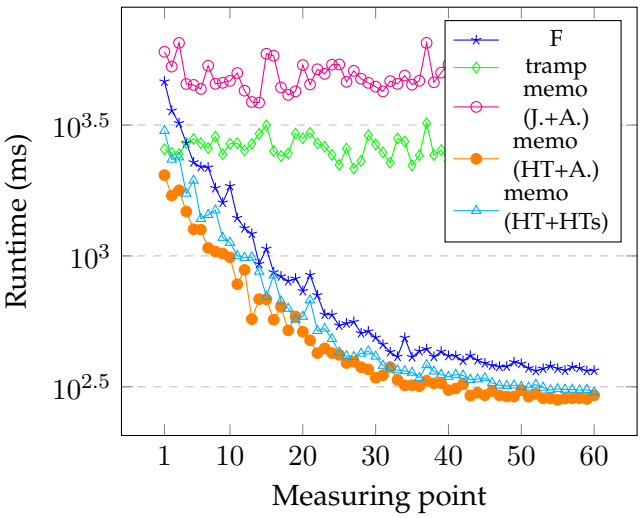

Figure 9: `comps`: runtime comparison w/ memoization.

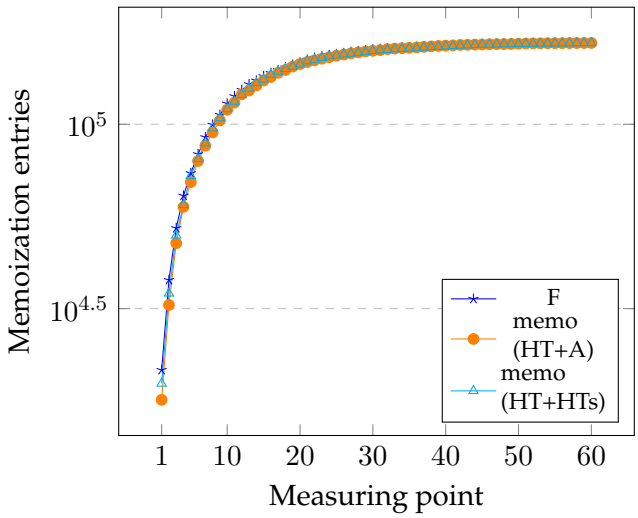

Figure 10: `comps`: comparison of memoization entries.

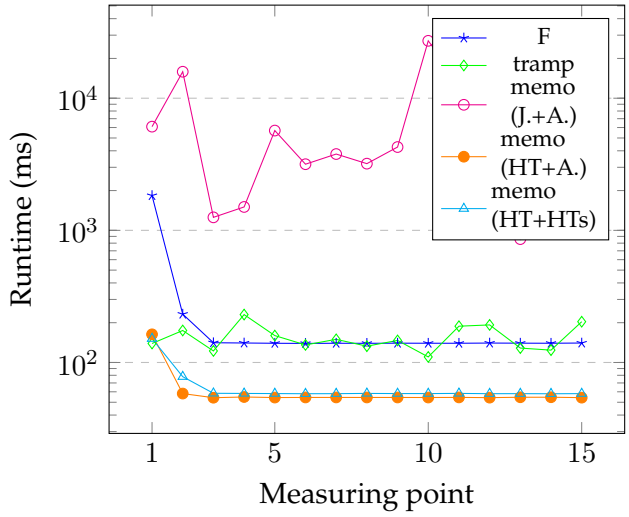

Figure 11: `eval`: runtime comparison w/ memoization.

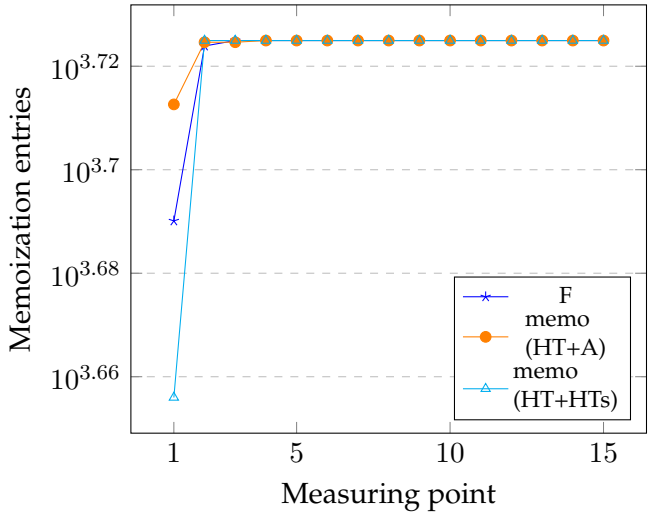

Figure 12: `eval`: comparison of memoization entries.

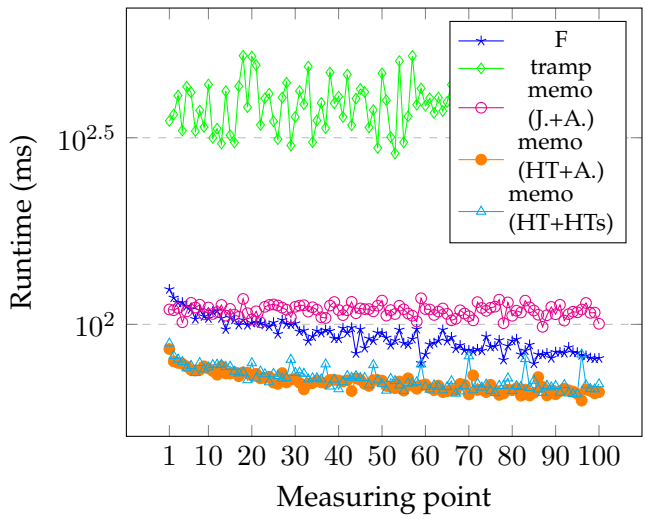

Figure 13: `lcs`: runtime comparison w/ memoization.

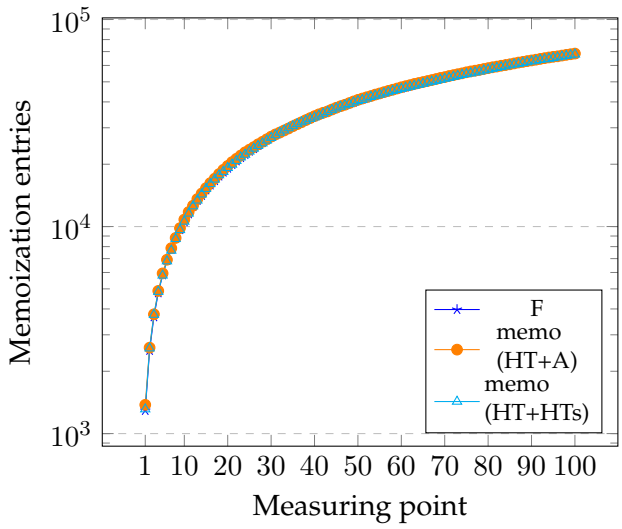

Figure 14: `lcs`: comparison of memoization entries.

List of Figures

Bibliography

[1] L. A. Rowe and M. R. Stonebraker, "The postgres data model," tech. rep., CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, 1987.

[2] C. Duta and T. Grust, "Functional-style sql udfs with a capital'f'," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1273–1287, 2020.

[3] T. P. G. D. Group, "Postgresql 9.1.24 documentation - chapter 7. queries." URL: https://www.postgresql.org/docs/9.1/queries-with.html. Accessed: 22.08.2021.

[4] D. Hirn and T. Grust, "One with recursive is worth many gotos," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 723–735, 2021.

[5] N. N. Vorobiev, *Fibonacci numbers*. Birkhäuser, 2012.

[6] GeeksforGeeks, "Geeksforgeeks - tail recursion." URL: https://www.geeksforgeeks.org/tail-recursion/. Accessed: 23.08.2021.

[7] P. S. (G-Research), "G-research - continuation-passing style." URL: https://www.gresearch.co.uk/article/continuation-passing-style/. Accessed: 24.08.2021.

[8] R. A. Kelsey, "A correspondence between continuation passing style and static single assignment form," *ACM SIGPLAN Notices*, vol. 30, no. 3, pp. 13–22, 1995.

[9] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 502–514, 2004.

[10] A. W. Appel, *Compiling with continuations*. Cambridge university press, 2007.

[11] O. Danvy and L. R. Nielsen, "Defunctionalization at work," in *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pp. 162–174, 2001.

[12] T. Johnsson, "Lambda lifting: Transforming programs to recursive equations," in *Conference on Functional programming languages and computer architecture*, pp. 190–203, Springer, 1985.

[13] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," *Higher-order and symbolic computation*, vol. 11, no. 4, pp. 363–397, 1998.

[14] B. Milewski, "Bartosz milewski's programming cafe - defunctionalization and freyd's theorem." URL: https://bartoszmilewski.com/2020/08/03/defunctionalization-and-freyds-theorem/. Accessed: 24.08.2021.

[15] S. E. Ganz, D. P. Friedman, and M. Wand, "Trampolined style," in *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pp. 18–27, 1999.

[16] T. P. G. D. Group, "Postgresql 9.4.26 documentation - chapter 18. server configuration," Accessed: 04.09.2021.

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Datum, Ort Unterschrift