EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

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

Masterthesis Computer Science

Extending WITH RECURSIVE With Multiple Working Tables And Additional Data Structures

Adrian Müller

31.03.2023

Examiner

Prof. Dr. Torsten Grust

Co-Examiner

Jun. Prof. Dr. Jonathan Brachthäuser

Supervisor

Denis Hirn

Adrian Müller:
Extending WITH RECURSIVE With Multiple Working Tables And Additional Data Structures
Masterthesis Computer Science
Eberhard Karls Universität
From 01.10.2022 to 31.03.2023

Selbständigkeitserklärung

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

Ort, Datum

Adrian Müller

Abstract

The *WITH RECURSIVE* construct in SQL as we know it is limited by providing only one working table to represent the state of the recursive CTE. In addition, whenever certain tuples generated throughout the recursive CTE are needed in more than just the next iteration, they must be actively copied. This thesis introduces modifications to standard PostgreSQL to allow multiple working tables, which may also store tuples consistently. Furthermore, even other data structures such as a stack, heap and hash table are implemented as tuple containers to be used instead of simple tables. The aim is to implement selected applications more efficiently by avoiding large numbers of copy operations and by providing faster access to specific tuples stored in the state of the recursive CTE.

Contents

Abstract v

Acronyms ix

1 Introduction 1

2 Implementation 5
- 2.1 Multiple *reset* and *extend* WTs . . . 5
- 2.2 Tuple Stack . . . 8
- 2.3 Tuple Heap . . . 12
- 2.4 Tuple Hash Table . . . 13

3 Application on chosen Examples 19
- 3.1 Graph Algorithms . . . 19
  - 3.1.1 Bellman-Ford . . . 19
  - 3.1.2 Dijkstra . . . 23
  - 3.1.3 A* . . . 26
- 3.2 Cursor Loop Application . . . 28
- 3.3 CPS Examples . . . 32
  - 3.3.1 Introduction . . . 32
  - 3.3.2 Closure Stack and multiple WTs . . . 35
  - 3.3.3 Memoization . . . 37

4 Measurements 43
- 4.1 Graph Algorithms . . . 43
- 4.2 Cursor Loop Application . . . 47
- 4.3 CPS examples . . . 48

5 Conclusion and Future Work 53

6 Appendix 57

Bibliography 59

Acronyms

CPS Continuation Passing Style
CTE Common Table Expression
IT Intermediate Table
PL/pgSQL Procedural Language/PostgreSQL
SFW SELECT FROM WHERE
SQL Structured Query Language
TS Trampolined Style
UDF User Defined Function
WT Working Table

1 Introduction

Today we live in a world where misleading and even downright false information seems to be thrown at us all the time, and the Internet provides a playground for its easy spread. Even PostgreSQL [1], a database management system and dialect for the Structured Query Language (SQL), is not safe from this. While this thesis will not try to start a scientific debate about the naming of the *WITH RECURSIVE* construct, I do feel obliged to point out that its naming is somewhat misleading. The *WITH RECURSIVE* construct extends a standard Common Table Expression (CTE), i.e. a temporary table that can be used within a query, *with recursion* - as the name implies. Hence, the *WITH RECURSIVE* construct is often also referred to as a recursive CTE. While the addition of the recursive CTE to SQL does indeed make SQL a Turing-complete programming language [2], calling the construct recursive might give the wrong impression. In fact, the construct is even implemented in an iterative rather than a recursive way. To understand this, consider Figure 1.1a, which shows a simple recursive CTE, and Figure 1.1b, which details the general workflow [3] for the *WITH RECURSIVE* construct.

```
1 WITH RECURSIVE t(x) AS (
2   SELECT 1             -- q0
3     UNION ALL
4   SELECT x+1 FROM t   -- qrec
5   WHERE  x < 5
6 ) TABLE t;
```

(a) Recursive CTE for the computation of the numbers 1 to 5

```
1 WT, UT = q0
2 WHILE WT ≠ ∅
3   IT = qrec(WT)
4   WT = IT
5   UT = UT ⊎ IT
6   IT = ∅
7 RETURN UT
```

(b) Workflow of the *WITH RECURSIVE* construct

Figure 1.1: Workflow for the *WITH RECURSIVE* construct demonstrated on a recursive CTE

Let us first look at the query using the *WITH RECURSIVE* construct, which defines the recursive CTE t. The recursion is as follows: The table t is first initialized by a query q_0, which in this case only computes one tuple with a column x valued 1. The result of the query q_0, in the case of the example query only the single tuple, is inserted into the Working Table (WT) [4] of the recursive CTE. This can also be seen in line 1 of the workflow in Figure 1.1b. The WT corresponds to a temporary table containing the current state of the recursive CTE t. The initialization of the WT is followed by multiple executions of the query q_{rec} over the course of several iterations. This query q_{rec} itself refers to the recursive CTE t, i.e. its current content is read and operated on. In the first iteration, the single tuple with $x = 1$ is read from the WT and a new tuple with an incremented x value of 2 is calculated. In general, the WT could have been initialized with more than one tuple, so the WT would be read multiple times. If the newly generated tuple were inserted directly into the WT, it would be read in the same iteration. However, only the previous state of the WT should matter for the computation of the new

state. Therefore, an Intermediate Table (IT) is defined, which is supposed to store the tuples generated during an iteration of the recursive CTE. In the workflow from Figure 1.1b, this corresponds to line 3. Only once an iteration has been completed and the next iteration is being prepared is the WT updated with the new tuples contained in the IT. This corresponds to line 4 of the workflow. Note that the previous contents of the WT are discarded, i.e. in the next iteration n only the tuples generated in iteration $n-1$ are available in the current state. However, the parent query following the recursive CTE is able to read all tuples generated throughout the recursive CTE, not just its final state. For this purpose, an additional union table UT is defined, which accumulates all tuples computed across all iterations. This can be seen in lines 1, 5 and finally in line 7, where the accumulated tuples are returned as the total content of the recursive CTE t. Therefore, the query shown in Figure 1.1a computes one tuple with an incremented x value in each iteration until the termination condition given by the *WHERE* clause is violated. This results in no new tuples being generated and thus an empty WT, which causes the loop in line 2 of the workflow to be exited.

Now that the workflow of the recursive CTE has been explained, it becomes clear that calling it a recursive construct is not quite fitting. It is not possible to issue some sort of recursive function call which, after having completed its calculations, would return to an outer context where the remaining calculations could be performed. While the naming of the recursive CTE as stated is not the subject of debate in this thesis, the restrictions on the *WITH RECURSIVE* construct certainly are. It is possible to use a functional approach to recursive CTEs by translating a function into the Continuation Passing Style (CPS), allowing it to pass the outer context as an additional argument to a recursive function call [5]. When also using defunctionalization and a translation into the Trampolined Style (TS), the function can then be realized by a recursive CTE. However, as will be shown in this thesis, this means that a possibly large outer context would have to be copied in each iteration of the recursive CTE, since the content of the WT is only available for exactly one iteration. This is also the case for when a function benefits from memoization, i.e. when the solution to a particular problem is computed by first solving smaller sub-problems whose results are needed several times for the top-level function call. All the data containing the solutions to the sub-problems would have to be copied in each iteration, which limits the benefits of memoization. There already is related work [6] that addresses this issue by allowing rows to be stored indefinitely throughout the recursive CTE, unless they are replaced by another row with the same key, or they expire by specifying a *time-to-live* value. The related work has already motivated the use of possibly many different WTs instead of just one, as is the case in standard PostgreSQL. Even the possible use of a stack data structure as an alternative to a common table as a tuple container has been hinted at. This thesis implements and tests exactly these proposals. Individual WTs may behave as usual, or they may accumulate tuples and prevent their contents from being discarded across iterations. This is already expected to eliminate the copying overhead for queries that benefit from memoization. An additional stack data structure may prove useful for CPS examples that pass their outer context to recursive calls. Even a currently researched translation from the more imperative Procedural Language/PostgreSQL (PL/PGSQL) to plain SQL [7] could benefit from the stack and allow an efficient SQL realization of cursor loops. If a stack data structure is already considered, then why stop there? A heap

data structure may lead to a better performance of certain graph algorithms such as Dijkstra and A* [8, 9]. These additional data structures allow the state of the recursive CTE to be stored over the course of several iterations, while allowing tuples to be read and removed according to their insertion order or a sorting criterion. A hash table using an upsert semantics similar to the previously mentioned related work [6] could also see its benefits, especially when allowing fast lookups of selected values. This thesis will detail the important implementation aspects of the new constructs, which extend the simple workflow of the recursive CTE presented above. The mentioned functions and queries will be implemented using the modified version of PostgreSQL and the difference in performance compared to standard PostgreSQL will be measured. Finally, the results are evaluated and an outlook for future work is given. The first chapter starts with the introduction to the implemented constructs.

2 Implementation

Now that the goal of this thesis has been addressed, the implementation of the new features for the recursive CTE will be presented. These will include the use of possibly many separate WTs, the introduction of an alternative update rule for *extend* WTs whose tuples are stored over multiple iterations, as well as a stack, heap, and hash table as additional data structures that may be used to store tuples instead of a standard WT. There are a number of customizations that need to be considered when implementing the specific features so that they can actually be used at the SQL-level. These design aspects and the constraints imposed on the features will be discussed in this chapter.

2.1 Multiple *reset* and *extend* WTs

First, the use of multiple WTs will be introduced, each of which can be declared separately as a *reset* WT or an *extend* WT. Regarding the use of multiple WTs, the goal is that an adapted *FROM* clause should determine the selected WT to be read. This selection should be done via a WT alias, i.e. the user can select a read operation on the WT with index 2 via a SELECT FROM WHERE (SFW) block containing the clause *FROM t as wt2*. Also, for each tuple generated by the recursive CTE, an additional *wt* column should now be mandatory, so that the tuple is inserted into the WT indexed by exactly that column. This column will contain the same *wt* index as the one used for the *wt* aliases. A query implementing these features is shown in Figure 2.1.

```
1 WITH RECURSIVE t(wt, x) AS (
2   SELECT 0, 1
3     UNION ALL
4   SELECT 0, x+1
5   FROM   t AS wt0r
6   WHERE  x < 5
7 ) TABLE t;
```

(a) Example query

wt	x
0	1
0	2
0	3
0	4
0	5

(b) Query output

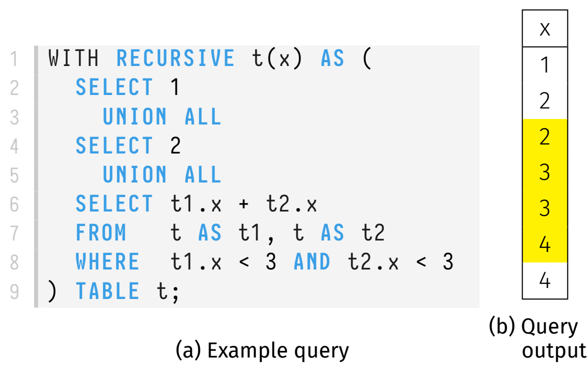
Figure 2.1: Example query demonstrating accesses to the WT with index 0

The scheme of the recursive CTE t itself already shows the additional *wt* column. Both the tuples generated in the non-recursive part as well as in the recursive part have the value 0 as a *wt* index, i.e. in both cases only the WT with index 0 is written to. Also, only this WT is read in each iteration, which can be deduced from the *FROM* clause containing the alias *wt0r*. In addition, each WT should have its own associated update rule. This determines how the WT should be modified at the end of the iteration. In standard PostgreSQL, the WT is usually overwritten by the tuples in the IT containing the intermediate results computed in the previous iteration. This now corresponds to the update rule

reset. An additional update rule, *extend*, now allows the contents of the IT to be simply added to the WT, while preserving the previous contents of the WT. This thesis will restrict itself to considering only *UNION ALL* linked SFW blocks, *UNION* clauses implementing a duplicate elimination will be ignored. The reason for this is that it only increases the implementation effort and does not add much insight for the research.

Let us now look at some of the relevant details of the announced features. We will begin by introducing the update rule *extend*. Using this, each tuple of the IT computed in the last iteration is added to the set of tuples contained in the WT. Unlike the standard update rule *reset*, tuples contained in the previous instance of the WT are preserved. In the following, we will concentrate on the distinction whether a given WT should be considered a *reset* or an *extend* WT. An obvious implementation would require each tuple to introduce yet another attribute to distinguish between a *reset* and an *extend* WT. In this case however, each tuple would already be extended by two additional attributes, namely the index for the target WT and the indication for the differentiation of the update rule. This means that the scheme of the recursive CTE is further inflated. To prevent this, it could be possible to allow only even WT indices for *reset* WTs and odd indices for *extend* WTs. However, it is far more useful to also implement the distinction via the WT alias. In order to do this, each WT is first assumed to use the *extend* update rule. This has the advantage that a WT that is not read at first may accumulate intermediate results. In a later iteration, when this WT is read for the first time, these intermediate results will still be present without the need to copy them in each iteration. Access to the tuples of a WT implementing the update rule *extend* is now possible by using the alias *wti* inside a *FROM* clause, where *i* refers to the index of the WT. Furthermore, by a read operation in the next iteration, it is possible to declare a WT as a *reset* WT. This can be done by using the alias *wtir* within a *FROM* clause, the additional *r* suffix will set the update rule to *reset*. Note that declaring the update rule *reset* one iteration after the corresponding WT has been filled with tuples is sufficient to mimic an immediate declaration of the update rule. For example, in iteration n a particular WT could be written to first, this WT will then use the default update rule *extend*. Since this WT would be read in the following iteration via an alias containing the *r* suffix, the tuples in the WT are discarded at the end of iteration $n+1$. Consequently, it seems as if the WT was a *reset* WT from the beginning.

Now the implementation of multiple WTs will be discussed in more detail. As explained in the motivation, the default implementation of the recursive CTE in PostgreSQL provides only one WT-IT pair. As a result, there is usually only one *WorkTableScanState* [10] instance during the execution of a recursive CTE. Across all iterations, this node is responsible for reading the tuples contained in the WT. We first need to make sure that multiple *WorkTableScanState* instances are allowed, so that we may read from different WTs later. In standard PostgreSQL, the amount of occurrences of the recursive CTE t are counted and this counter is incremented for each row variable in a *FROM* clause bound to t. Normally, this counter must be exactly 1, as a counter of 0 would infer a non-recursive CTE. Multiple reads of the recursive CTE are also prevented by throwing an error. However, this can be avoided by simply uncommenting the counter check in the code. This makes it possible to access the recursive CTE t through multiple row variables within one or more *FROM* clauses, as shown in Figure 2.2.



```sql
WITH RECURSIVE t(x) AS (
  SELECT 1
    UNION ALL
  SELECT 2
    UNION ALL
  SELECT t1.x + t2.x
  FROM   t AS t1, t AS t2
  WHERE  t1.x < 3 AND t2.x < 3
) TABLE t;
```

(a) Example query

x
1
2
2
3
3
4
4

(b) Query output

Figure 2.2: Example query computing the join of a WT with itself

Here, two row variables t_1 and t_2 are bound to the contents of the recursive CTE t, so two separate *WorkTableScanState* instances are created for reading t. After t is initialized with the tuples containing the x values 1 and 2, the first iteration computes the join of t with itself. For the result of the join, which would be the Cartesian product of the contents of t, we would expect four generated tuples with x values $2, 3, 3$ and 4. These tuples computed in the first iteration of the recursive CTE are highlighted in yellow in the query output from Figure 2.2b. However, since usually only one *WorkTableScanState* instance is used, the two row variables t_1 and t_2 and thus the *WorkTableScanState* instances still share their read pointers, rather than having individual ones. When the row variable t_1 accesses the first tuple 1 of the WT, the read pointer is incremented. As a result, the following access to the same WT by the row variable t_2 would read the next tuple 2 instead of also reading the first one. This means that only a tuple with the x value 3 would be computed in the first iteration. A second iteration would not compute any new tuples at all, since there is only one tuple in the WT to be read for only one of the row variables. To fix this, and to allow an actual join of the set of tuples with itself, the *WorkTableScanState* definition needs to introduce a unique read pointer. Now that each instance has its own read pointer, it is possible for each row variable to iterate over the tuples of a WT completely independently of other row variables. Finally, each *WorkTableScanState* instance should also store the index of the assigned WT, as each instance or row variable should only be able to read exactly one WT. The assignment of the WT index is handled by a function for interpreting the alias *wti[r]*, which is present in a *FROM* clause *FROM t AS wti[r]* of a SFW block. This function also detects a potential *r* suffix in the alias and sets the update rule for the WT accordingly. It is noteworthy that the join of a WT with itself is still possible. For example, we could choose two aliases *wt1a* and *wt1b*, meaning the same WT 1 would be read by different *WorkTableScanState* instances. Now it is possible to introduce multiple WTs. Instead of just one WT-IT pair, we now define one array each for the WTs and the ITs. Additionally, a third array must be provided for the update rules, which determines for each WT-IT pair whether it should be updated via the *extend* or the *reset* rule. Initially, only one WT-IT pair is defined, so the arrays contain only one element and only the WT index 0 exists. If a tuple is computed during the computation of the recursive CTE, or alternatively a row variable with a particular alias is defined in a *FROM* clause, one of which may hold a new WT index, another WT-IT pair and an update rule will be dynamically allocated. If the number of required WT-IT pairs exceeds the number of fields available for both the WT and IT arrays, new arrays with twice as many fields are allocated and the WTs and ITs are copied into the new arrays. Since the individual WTs and ITs are only referenced by their pointers, we only need to copy these pointers rather than the contents of

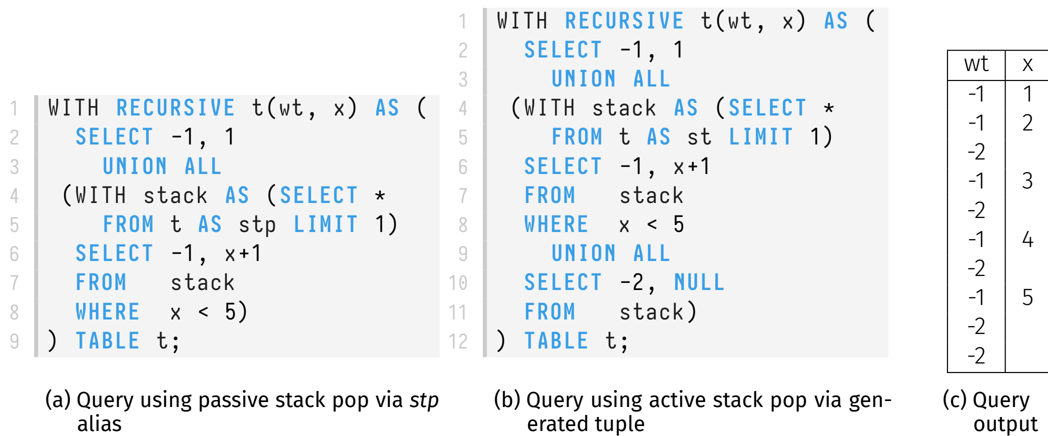
the tables, making this dynamic behaviour feasible.

With these modifications to standard PostgreSQL, it is now possible to use any number of individual WTs. Each WT may accumulate tuples over the course of many iterations by using the new update rule *extend*. With this it is possible, for example, to store the memoization contents throughout the whole computation performed by the recursive CTE without having to copy the memoization tuples in each iteration. In the later chapter 4, several queries will be executed and measured, which will benefit from the usage of memoization, especially by using a separate memoization WT. Before we look at optimizing existing queries, the remaining data structures are going be introduced. The next chapter introduces the first of these data structures, the tuple stack.

2.2 Tuple Stack

Using the examples in the CPS and the TS as the main motivation for this thesis, yet another optimization can be identified: The examples in the CPS all use a stack to store the closures, which will be demonstrated in the later chapter 3.3.2. In the given queries, however, the closure stack is implemented by an array. One problem with using an array is that by only performing a maximum of one stack pop per iteration, the closure stack has to be copied over many iterations. A more desirable approach would be to use another update rule similar to the *extend* rule, where the closure stack can be kept alive during the course of the recursive CTE. First of all, the closures should now be atomic, meaning each closure is now represented by a single tuple. We also need to consider the order of the tuples on the stack as well as pop operations, so the update rule *extend* is not sufficient. Instead, an actual tuple stack is implemented for the storage of the tuples. In this chapter, I will clarify certain relevant aspects for all data structures, as well as the detailed ones for the stack. The difficulty in implementing the different data structures lies in the following points: It needs to be determined how push and pop operations may be possible at the SQL-level. We might also want to allow simple read operations rather than just pop operations. Certain state changes in a data structure itself through push or pop operations would also have to result in another iteration being performed. It is also important to consider a certain order for the operations performed on the data structure within an iteration: At the start of the iteration, the data structure should be read and possibly pop operations should be performed. Only then should push operations have an effect on the actual data structure, otherwise we may experience an unwanted behaviour. For example, in the case of the stack, a tuple which was just pushed onto the stack might be discarded immediately if a pop operation were performed within the same iteration after the stack push. All of these difficulties will be addressed below.

Let us start with the syntax in which the stack is used within a *WITH RECURSIVE* SQL query. For the most part, I will follow an approach similar to the one of multiple WTs. There, the additional *wt* column of a tuple generated by the recursive CTE determines the WT in which the tuple is to be inserted. Since currently only WT indices corresponding to an array index are allowed - i.e. only positive integers starting from 0 - negative integers are still available. Thus a value of



```sql
WITH RECURSIVE t(wt, x) AS (
  SELECT -1, 1
    UNION ALL
 (WITH stack AS (SELECT *
    FROM t AS stp LIMIT 1)
  SELECT -1, x+1
  FROM   stack
  WHERE  x < 5)
) TABLE t;
```

(a) Query using passive stack pop via *stp* alias

```sql
WITH RECURSIVE t(wt, x) AS (
  SELECT -1, 1
    UNION ALL
 (WITH stack AS (SELECT *
    FROM t AS st LIMIT 1)
  SELECT -1, x+1
  FROM   stack
  WHERE  x < 5
    UNION ALL
  SELECT -2, NULL
  FROM   stack)
) TABLE t;
```

(b) Query using active stack pop via generated tuple

wt	x
-1	1
-1	2
-2	
-1	3
-2	
-1	4
-2	
-1	5
-2	
-2	

(c) Query output

Figure 2.3: Example queries showing the use of the tuple stack

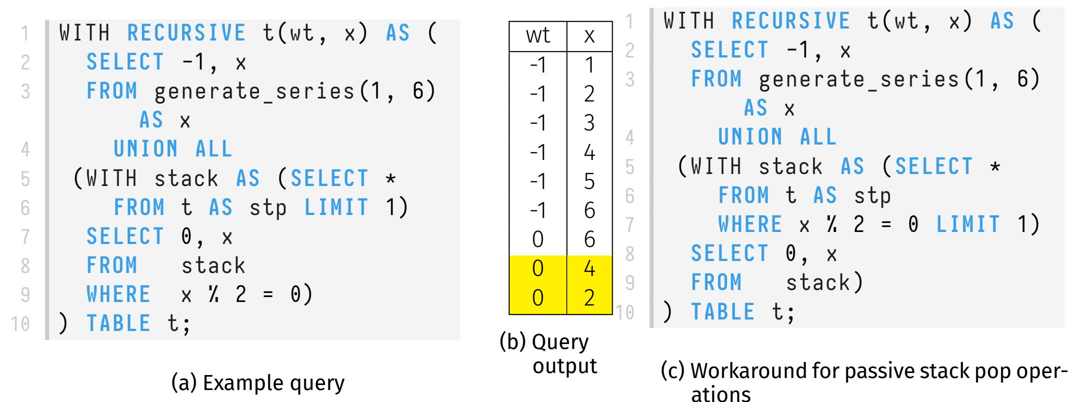
−1 in the *wt* column will now refer to a stack push. Also, reading tuples from the stack is now implemented using a special alias. So, for simple tuple reads, the alias in a *FROM* clause should be *st*. Reads are useful for the stack because some CPS queries, which will be presented later, often access a tuple of the closure stack without simultaneously removing the read tuple by a pop operation. Besides a read from the stack using the alias *st*, another read including a pop operation is implemented by selecting the alias *stp*. By using this alias, the topmost tuple will be read again, but it will also immediately undergo a pop operation, meaning that any tuple read will also be removed from the stack. The use of the alias *stp* is especially useful for the cursor loop example shown in the later chapter 3.2, as the tuples read from the stack are no longer needed. The alternative to a stack pop via an alias is a bit more cumbersome: It is also possible to declare a stack pop by generating a tuple with a *wt* column valued at −2. This makes it possible to define certain conditions within the corresponding *WITH* clause, so that a pop should only occur if the condition is fulfilled. This alternative way of implementing a stack pop is useful for the examples using the *st* alias, as there still needs to be a way to perform pop operations. Note that the simple stack read via the alias *st* and, as a consequence, the stack pop via a generated tuple are optional and only implemented for convenience. Two simple queries using the tuple stack are shown in Figure 2.3.

The initial focus should be on the read, pop and push operations through generated tuples and the use of a particular stack alias. The purpose of the *stack* CTE will be explained shortly. Both the queries are equivalent in terms of the stack, only the implementation is different. In the non-recursive part of the query, the stack is initialized with one tuple. This is followed by the recursive part, which reads the contents of the stack and, under a certain condition, performs another stack push. In the query 2.3a, the read tuple will immediately be removed from the stack by using the alias *stp*. In the query 2.3b, this is done by actively computing a tuple with the value −2 in the *wt* column. This means that in almost every iteration both one push and one pop operation are performed, as it can be seen in the query output in Figure 2.3c. Note that the

tuples with a *wt* column valued with −2 are only generated for the query 2.3b.

Now that the implementation of the relevant operations has been presented, the order in which these operations will be carried out needs to be be established. First, since pop operations always occur as soon as the alias *stp* binds a row variable to the topmost tuple, or a tuple with the value −2 in the *wt* column is generated, the stack instantly experiences a state change. So, read operations must occur before pop operations, otherwise tuples could be removed from the stack before being read. This can be ensured by using a centralized *stack* CTE within the recursive CTE, so that all stack reads occur right at the very start of an iteration. In addition, all SFW blocks that read from the stack may now do so by accessing the previously computed *stack* CTE. To ensure that the CTEs are actually computed before the following parent query, the CTEs may need to be defined with the additional specification of *AS MATERIALIZED* [11]. This prevents a potential SQL optimization by the planner [12] where the CTE could be inlined into the parent query if there is only one occurrence of the CTE. However, it is more difficult to manage the order for push operations. This can already be seen in query 2.3b, where both the stack push and the pop are instructed by generated tuples. We cannot be sure that pop operations will always be performed before push operations because of potential SQL optimizations that could change the order in which query parts are executed. In the example query 2.3b, an unwanted behaviour could occur where the tuple that was just pushed to the stack is immediately removed by the following pop operation. So, we need ensure sure that stack pushes are always performed last in some other way: When a tuple with a *wt* column of −1 is computed, which would have to be written to the stack, the tuple is first cached. Only when the current iteration has terminated and the next iteration is being prepared should the tuple actually be stored on the stack. By then, all pop operations will have been performed, regardless how they happened. Within an iteration, it is generally allowed to push multiple tuples to the stack which requires the need to cache possibly many tuples. For this reason, the stack was implemented as a singly linked list. First, the stack itself is represented as a list, but so it the data structure used to cache the tuples to be pushed at the end of the iteration. At the end of the iteration, the actual stack pushes may then be performed by simply concatenating these lists. For an array implementation of the stack, this would be less elegant and much more expensive. It has just been established that multiple stack pushes per iteration are allowed. Even the insertion order of simultaneously inserted tuples can be specified by the user with an *ORDER BY* clause. In addition to multiple stack pushes per iteration, many read and pop operations could also be performed. This is indicated by an additional *LIMIT n* clause within the *stack* CTE, which instructs a total of n stack reads. Note that this only makes sense when using the *stp* alias, as simple reads using the *st* alias would only read the topmost tuples multiple times. So, not providing a *LIMIT* clause at all would result in an infinite loop in the case of a *st* alias. However, using a *stp* alias without a *LIMIT* clause would simply cause all the tuples on the stack to be read and removed. A *LIMIT* clause should therefore always be provided.

There is still a design aspect to be addressed, concerning both the stack and the heap data structures: In standard PostgreSQL a new iteration is invoked whenever at least one tuple has been inserted into the IT during the iteration, i.e. whenever the recursive CTE has computed a tuple and a state change has occurred. The modified implementation with multiple WTs instead



```sql
WITH RECURSIVE t(wt, x) AS (
  SELECT -1, x
  FROM generate_series(1, 6)
      AS x
    UNION ALL
 (WITH stack AS (SELECT *
    FROM t AS stp LIMIT 1)
  SELECT 0, x
  FROM   stack
  WHERE  x % 2 = 0)
) TABLE t;
```

(a) Example query

wt	x
-1	1
-1	2
-1	3
-1	4
-1	5
-1	6
0	6
0	4
0	2

(b) Query output

```sql
WITH RECURSIVE t(wt, x) AS (
  SELECT -1, x
  FROM generate_series(1, 6)
      AS x
    UNION ALL
 (WITH stack AS (SELECT *
    FROM t AS stp
    WHERE x % 2 = 0 LIMIT 1)
  SELECT 0, x
  FROM   stack)
) TABLE t;
```

(c) Workaround for passive stack pop operations

Figure 2.4: Example query showing different results when passive stack pop operations are or are not counted as state changes

performs another iteration if at least one IT has been written to. Now that additional data structures are implemented, any kind of state change in these data structures should generally also trigger another iteration. This includes any stack push or pop. In the case of stack pushes or pops through a generated tuple with a *wt* column valued –1 or –2 respectively, this is obvious, as a tuple has been actively computed. Only with stack pops via a *stp* alias is it questionable whether these should also start another iteration if no tuple was computed during the iteration. Strictly speaking, it should actually trigger another iteration because a state change has occurred, which could result in more computations being done in the next iteration. An example for this is given in Figure 2.4a.

Here, a tuple is only computed in the recursive part of the query if the tuple read from the stack has an even x value. If a passive stack pop via alias *stp* were able to start a new iteration on its own, every tuple from the stack that satisfies the condition would be used to compute an output tuple. In contrast, if only actively computed tuples were considered for state changes, the query would terminate after reading a stack tuple with an odd x value. This means that the tuples highlighted in yellow in the query output of Figure 2.4b are only computed by the query of Figure 2.4a if a passive stack pop via the alias *stp* would count as a state change. In general, it may happen that the result of the recursive CTE has already been computed, but the data structure still contains tuples. In this case, many more useless iterations would be performed without any generated tuples until the data structure is actually empty. An example of this can be seen later in the tuple heap data structure, especially for an A* function call where the result may be computed within just a few iterations, but the heap would still contain many more tuples. The negative impact of counting stack pops through the *stp* alias as state changes that trigger another iteration led me to not count them as state changes. So, only actively computed tuples can lead to another iteration, while passive stack pops via a *stp* alias do not. This problem can also be avoided by adding a *WHERE* clause to restrict the tuples read from the data structure. An example of this is shown in Figure 2.4c.

The *stack* CTE will now ensure that a contained tuple fulfills the given condition. If the topmost tuple itself does not meet the condition, it is still removed from the stack via a passive pop operation and the new topmost tuple is considered. If none of the tuples on the stack satisfy the condition, they are all removed from the stack within one iteration. This way of handling pop operations from a data structure is also retained for the tuple heap. The next chapter takes a closer look at the heap.

2.3 Tuple Heap

The tuple heap offers another use case, namely easy access to a tuple with an extreme value via a priority queue. It was implemented using an array, so that both heap inserts as well as deletes could benefit from a complexity of $O(\log n)$ [13]. The two graph algorithms Dijkstra and A* were chosen as two areas of application for the heap, they are described in detail in the chapter 3.1. Both algorithms are based on the same principle, i.e. within an iteration, the node of a set of nodes with the lowest cost value is estimated and expanded. In practice, both algorithms are usually implemented using a priority queue, as this has the advantage of storing the set of nodes in an ordered fashion according to their cost value. The next node to expand from can therefore be read directly from the root of the heap with a complexity of $O(1)$. However, the node must also be removed from the set of nodes and thus the heap property must be restored, for a total effort of $O(\log n)$. By comparison, implementing the set of nodes via a normal WT would require an effort of $O(n)$ to read the next node to expand from, as the entire WT would have to be searched. Insertions into the heap would then also only have a lower complexity of $O(1)$, but it would be necessary to copy almost the whole WT in each iteration, which is why the running time of a query implementing a WT for the set of nodes is expected to be much higher. The update rule *extend* cannot be used because expanded nodes should be removed from the WT and this is not possible in the current implementation of the WT.

The implementation of the heap is very similar to the one of the stack. Heap insertions, also called enqueue operations, are now triggered by generated tuples containing a *wt* column with value -3. The heap may again be read via an alias in a *FROM* clause, this alias should be *he*. Each read operation corresponds to an actual dequeue operation, i.e. the root element of the heap is first read, then removed from the heap, and finally the heap property of the heap is restored. Simple read operations without a dequeue operation are not implemented for the heap, as they seem unnecessary. Again, it is possible to write multiple tuples to the heap per iteration, and multiple reads are also allowed by using a *LIMIT* clause. Analogous to the stack, the heap should be accessed via a centralized *heap* CTE, so that heap reads, and thus dequeues, are consistently performed before enqueues. Since the tuples are stored in an ordered fashion with respect to a cost value, it is necessary to specify which column of a tuple should contain the cost value. Since the first column of the tuple is still reserved for the *wt* column, the second column was chosen for this. To use the heap, this column should now always contain an integer value. Since both Dijkstra and A* require a min heap, the root of the heap will always store the tuple with the lowest value in the second column. A max heap could still be implemented by

```sql
WITH RECURSIVE t(wt, x, iter) AS (
  SELECT -3, (random() * 100) :: int, NULL :: int
  FROM   generate_series(1, 100) AS x
    UNION ALL
  SELECT 0, NULL :: int, 0
  UNION ALL
 (WITH heap AS (SELECT * FROM t AS he LIMIT 1)
  SELECT 0, heap.x, wt0r.iter + 1
  FROM   heap, t AS wt0r)
) SELECT * FROM t WHERE wt = 0;
```

(a) Example query

wt	x	iter
0		0
0	2	1
0	2	2
0	2	3
0	2	4
0	5	5
0	6	6
0	6	7
0	7	8
0	7	9
0	7	10
...	...	...

(b) Query output

Figure 2.5: Heap sort implemented using the tuple heap

either multiplying all cost values by -1, or a clean implementation could be easily added at the C-level. As this is not of interest for the implemented examples, it has been omitted. Overall, the implementation of the heap is simpler than the one of the stack, because the order of the operations is already given by the use of a *heap* CTE and by not allowing simple read operations. As a consequence, we don't need an additional data structure to cache tuples to be added to the heap later, enqueue operations can be performed right after the computation of a generated tuple with a *wt* column with value -3. Finally, Figure 2.5 shows an example query to demonstrate the syntax introduced. In the non-recursive part of the query, 100 tuples with random numbers in the interval $[0,100]$ are stored on the heap. The random number is placed in the second column of the tuple, and thus serves as the ordering criterion for the priority queue. In each iteration, the root element of the heap is read and an incremented *iter* column is computed, so that the result shows which x value was calculated in which iteration. The query reflects heap sort, as with each incremented *iter* column, the tuple with the next highest *x* value is given.

With the implementation of the stack and the heap on top of multiple WTs, we now have two additional data structures in our arsenal. Another will be added in the next chapter, the hash table.

2.4 Tuple Hash Table

While the priority queue just explained is only useful for a specific selection of problems, the hash table is more accessible and useful more often. The implementation of the hash table means that, for the first time, indexing support [14, 15] is available within a recursive CTE. For normal WTs, the order of the tuples with respect to a particular attribute is not maintained, so that the search for a tuple with a particular attribute value would have to consider the whole WT. A hash table, on the other hand, allows an indexed lookup for a given key attribute, where only one of possibly many buckets - that is, a small subset of the full set of tuples - needs to be considered. To make the hash table compatible with SQL, it has to be adapted in many ways.

First, the upsert operation [6] for associating a key attribute with a tuple will be introduced. The upsert contains both an update as well as an insert operation, hence the name upsert. An upsert into the hash table can be requested by computing a tuple with the value -4 in the *wt* column. The key of the tuple to be associated should always be given as an integer value in the third column of the tuple. Depending on the key value and the total number of buckets available for the hash table, a hash value is calculated and the tuple is then inserted into the bucket indexed by that hash value. This is the simple insert part of the upsert operation. The hash table itself is implemented as an array of singly linked lists, where each array field corresponds to one bucket and thus to one of the lists. This means that if you need to look up a particular key attribute later, you only need to search one of the lists with a subset of the tuples, rather than the whole data structure, if index support is missing. Also relevant is the second column of the associated tuple which should contain an integer or numeric value. This second column is important for the update part of the upsert operation, which occurs when there is already an association present in the hash table for the key to be associated with another tuple. These conflicting associations for a key value need to be resolved appropriately. In programming languages such as C, the sequential behaviour of the program makes it easy to control updates to hash table or dictionary entries. Again, this is not possible in SQL because we cannot control the order of computations within an iteration. This means that if we compute multiple conflicting upsert tuples for a key value within an iteration, the tuple kept would be chosen arbitrarily by SQL, depending only on the order of execution, which we have no direct control over. We solve this by using the second column of an upsert tuple as the comparison value. For each key value, only one association is allowed in the hash table, and for conflicting tuples with the same key value, the tuple with the lower comparison value is retained. With this change, the hash table is now compatible with the heap, as both require an ordering criterion in the second column of a tuple, which is optimized for the lower values. This will make it possible to use both the heap and the hash table to further improve A* and Dijkstra queries. For now, a simpler example of the features explained is given in Figure 2.6.

In the non recursive part of the given query, multiple upsert operations are performed for conflicting tuples with a common shared key value 1. According to the result of the query read into WT 1, only the tuple $(-4, 3, 1)$ is kept as the association for the key 1, since its comparison value 3 is the lowest of the conflicting tuples. In addition, a delete operation for the removal of the association of the key with its tuple is demonstrated. Such a delete operation may be performed by generating a tuple with a *wt* column with value -5. Analogous to the upsert operation, the delete tuple should contain the key value in the third column. If the hash table actually contains an association with the given key value, that entry is discarded from the hash table. The delete operation will not be used in the queries measured in the later chapter 4. However, having the option to use it only makes the implemented hash table richer, and it is safe to say that implementing a delete for the hash table is easily done.

The upsert and delete operations introduced two variants that allow the hash table to be modified. Now the methods for reading the contents of the hash table will be presented. The simpler method allows all the tuples of the hash table to be read. As with all the constructs shown previously, this is again done via an alias within a *FROM* clause of the recursive CTE. The

```sql
WITH RECURSIVE t(wt, x, key) AS (
  SELECT -4, 5, 1
    UNION ALL
  SELECT -4, 3, 1
    UNION ALL
  SELECT -4, 8, 1
    UNION ALL
 (WITH hashtable AS (SELECT * FROM t AS ht)
  SELECT 1, ht.x, ht.key
  FROM   hashtable AS ht
    UNION ALL
  SELECT -5, NULL, ht.key
  FROM   hashtable AS ht)
) TABLE t;
```

(a) Example query

wt	x	key
-4	5	1
-4	3	1
-4	8	1
1	3	1
-5		1

(b) Query output

Figure 2.6: Demonstration of upserts, deletes and reads using the tuple hash table

alias *ht* has been chosen for this and, as before, it is recommended to read the hash table in a custom CTE at the start of an iteration. This is also shown in the query from Figure 2.6. A more interesting method of reading from the hash table is to take advantage of the index support via a lookup. The goal, as mentioned earlier, is to provide a key value to read the associated tuple, which may be found faster because we only have to consider one of the many buckets. However, with the constructs available in SQL, this is difficult to implement. In principle, it would be possible to extend the alias for the hash table with the key that would be read at the C-level, and depending on that, the indexed tuple could be bound to the row variable given by the alias. Unfortunately, aliases cannot be computed, so the key for the lookup would have to be hard-coded. This is why this option is not considered, as it makes the lookup really inflexible. An alternative approach would be to first generate a tuple containing a key value, which provokes a state change in the hash table storing that key value. A subsequent read of the hash table using a lookup alias could then use the stored key value as the lookup key. The disadvantage of this approach is that it would require two SFW blocks to be used for each lookup, which would also have to be executed in strict order. Again, SQL optimizations that might alter the order of the execution of SFW blocks would be problematic, so the two SFW blocks would have to be evaluated in one iteration each. This further reduces the usefulness of the hash table, especially since it would only be possible to look up a single key value every other iteration. Therefore, the only feasible option considered was to use a system information function [16]. Again, this has its own limitations: A system information function can only return a multi-valued record type if its exact type is predefined. However, we want to allow the return of an arbitrarily typed tuple, so this is no longer an option and we need to only implement the lookup of a single tuple attribute. Since the main purpose of this thesis is to investigate the usefulness of the implemented data structures for recursive CTEs, many of the limitations of the hash table are tolerated. All of the modifications of standard PostgreSQL made in this thesis are based on hacks that extend the existing syntax constructs with additional semantics. If these additional features were to be

```sql
WITH RECURSIVE t(wt, x, key) AS (
  SELECT 0, 0, NULL
    UNION ALL
  SELECT -4, x, ROUND(random() * 6) :: int
  FROM   generate_series(1, 3) AS x
    UNION ALL
  SELECT 0, x+1, hashtablelookup(2, x+1) :: int
  FROM   t AS wt0r
  WHERE  x < 5
) TABLE t;
```

(a) Example query

wt	x	key
0	0	
-4	1	3
-4	2	1
-4	3	5
0	1	2
0	2	-1
0	3	1
0	4	-1
0	5	3

(b) Query output

Figure 2.7: Demonstration of the system information function *hashtablelookup*

added to a PostgreSQL installation for a production environment, additional syntax elements should be included to make the features easier and more intuitive to use. Propositions for these additional syntax elements will be given in the conclusion of this thesis. If these specific syntax elements were actually present, a better lookup could easily be implemented without the detour using a restrictive system information function.

Now let us continue, the lookup is only allowed to read one attribute of the associated tuple. To do this, we use two parameters for the system information function *hashtablelookup*: The second parameter specifies the key value for the lookup of the tuple associated with that exact key. The first parameter is the index of the column to be read. In this case the indexing will start with 1. It is quite unnecessary to estimate the *wt* column per lookup, which will always be -4 for any associated tuple anyway, as well as the key value that has to be given as an argument to the lookup to begin with. So, the column indices 1 and 3 should be avoided. For the actual implementation the lookup for these column indices is not even possible, only the lookup for the column indices 2 and 4 are allowed. The reason for this is that while it was possible to access certain indexed attributes from a tuple computed by the recursive CTE, it was not possible to access indexed attributes from the copy of the tuple in the hash table. The example queries shown later, which actually use hash table lookups, only require a maximum of two lookup columns. Therefore, only the tuple attributes with column indices 2 and 4 are extracted from an upsert tuple and these will be stored in additional attributes of the association. This could certainly be improved by ensuring that the copy operations are fully functional, but for the purposes of the research this limited implementation should suffice. In the example queries shown later in the 3 chapter, hash table lookups are often performed throughout the iteration rather than via CTEs, which strictly occur before the following SFW blocks. This makes it necessary to cache all the upsert and delete tuples generated throughout the iteration, so that the lookups can only read the state of the hash table from the previous iteration. As with the stack pushes, the actual upsert and delete operations are only performed at the end of the iteration. This does not lead to any meaningful overhead, because for each hash table entry, a struct instance is defined that contains the copied tuple. This struct instance is referenced by a pointer, so performing the actual upsert or delete operation does not come at a high cost. Finally, an example query

for using a lookup is now shown in Figure 2.7. In the non-recursive part of the query, 3 tuples with random key values in the interval $[0, 5]$ are upserted into the hash table. In the recursive part, lookups are performed over multiple iterations for all key values in the same interval. For each association present in the hash table, the comparison value from the second column is returned, while the error value -1 is returned for non-existent associations.

Now all the implemented constructs were presented. These include the use of multiple WTs, each using either the new default *extend* update rule, or alternatively the *reset* update rule. In addition, a stack, heap and hash table instance have been introduced, which should benefit queries using ordered access to tuples. All new features allow tuples to be stored across multiple iterations, avoiding potentially many copy operations for tuples needed in more than one iteration. The following chapter will present examples and their SQL implementations, which will be optimized for runtime by using the new constructs. As a summary and as an aid to checking the features, Listing 2.1 shows the form used by the different constructs.

```
1  WITH RECURSIVE t(wt, <scheme>) AS (
2    // non-recursive part
3    <SFW block> [ UNION ALL <SFW block> ]*
4      UNION ALL
5    // recursive part
6   ( [ WITH <name> AS (<SFW block>) [ , <name> AS (<SFW block>) ]* ]
7    <SFW block> [ UNION ALL <SFW block> ]*
8    )
9  ) TABLE t;

10 <SFW block> :=
11   SELECT <selection>
12   FROM   t AS <alias>
13   WHERE  <condition>
14   LIMIT  <n>

15 <selection> :=
16   (i,  ...)   //  i in [0, inf): wt insertion
17   (-1, ...)   //  stack push
18   (-2, ...)   //  stack pop
19   (-3, <ordering-criterion>, ...)        // heap insert
20   (-4, <ordering-criterion>, <key>, ...) // hash table upsert
21   (-5, NULL, <key>, ...)                 // hash table delete

22 <alias> :=
23   wti   // i in [0, inf): wt read, set update rule to extend
24   wtir  // i in [0, inf): wt read, set update rule to reset
25   st    // stack read
26   stp   // stack read & pop
27   he    // heap  read & pop
28   ht    // hash table read

29 <hash table lookup> :=
30   hashtablelookup(<column>, <key>)   // <column> in {2, 4}
```

Listing 2.1: Summary of the introduced constructs

3 Application on chosen Examples

In this chapter, many sample queries will be presented and modified using the new constructs. The actual measurements of the modified and hopefully optimized queries will be evaluated in the later chapter 4. The examples chosen include many queries that are already implemented in the CPS and TS and for which the optimizing potential is quite high. The optimizations will include the use of multiple WTs to implement the trampolization more intuitively, an *extend* WT to hold memoization contents over many iterations, and a closure stack to replace the inefficient closure array. I will also try to draw profit from a hash table for memoization purposes. To further demonstrate the usefulness of the stack, a cursor loop application from the more imperative PL/PGSQL will be realized with pure SQL. In addition, the graph algorithms A* and Dijkstra were chosen as applications for the heap. For both of these, as well as a third graph algorithm in Bellman-Ford, the hash table will also play a role. The next chapter will begin with the graph algorithms just announced.

3.1 Graph Algorithms

All three chosen graph algorithms have one property in common: They all aim to compute the cheapest path from a single starting node s to one or more other nodes of a weighted and directed graph. The cheapest graph to an end node e corresponds to a set of nodes $\{s, ..., e\}$, for which every neighbouring pair of nodes (u, v) must exist as an edge in the *edges* table representing the graph. This table stores all the edges containing the start and destination nodes, as well as the cost or weight required to take that edge. The cheapest path is therefore the set of nodes whose summed up weights of all edges connecting the nodes is minimal. The algorithms differ in terms of different constraints on the graph or the presence of additional information. It is important to understand the algorithms, at least superficially, so that the possible optimizations may be estimated accordingly. The first part focuses on the implementation and optimization of the Bellman-Ford algorithm.

3.1.1 Bellman-Ford

The Bellman-Ford algorithm [17] computes the cheapest paths from a starting node s to every other node in the graph. To achieve this, for each node v the cheapest possible total distance $dist$ from the starting node is estimated, as well as the last-hop node $prev$ from which v is finally reachable by edge (u, v). The pseudocode [18] is shown in Figure 3.1.

Figure 3.1: Bellman-Ford pseudocode

Algorithm 1 Bellman-Ford

```
1:  for i ← 1 to n do
2:      dist[i] ← ∞
3:  end for
4:  dist[s] ← 0
5:
6:  for i ← 1 to n − 1 do
7:      for each (u,v) ∈ E do
8:          if dist[v] > dist[u] + weight(u,v) then
9:              dist[v] ← dist[u] + weight(u,v)
10:             prev[v] ← u
11:         end if
12:     end for
13: end for
```

At the start in lines 1-3, the distance values *dist* for all nodes are initialized with an infinite value, only the starting node is immediately reachable at a cost of 0. After initialization, the updates for the distance values and the last-hop nodes take place in lines 6-13. This is done over several iterations where each edge (u, v) is checked. If the node v is cheaper to reach using this edge, both the distance value and the last-hop node of v are updated. A special feature of the Bellman-Ford algorithm is that negative weights are allowed for edges. Only negative cycles must not appear in the paths computed by the algorithm. A negative cycle corresponds to a path from a node to itself that can be traversed by following a set of edges with an accumulated negative weight. If negative cycles were allowed, the negative cycle could be repeated an infinite number of times. So, any node reachable from any of the nodes in the cycle would theoretically be reachable from the starting node at a costs of $-\infty$. To prevent this, an additional iteration of cost updates can be performed to implement negative cycle detection. Now that the Bellman-Ford algorithm has been roughly explained, the SQL implementation will be presented. Consider the query in Figure 3.2, which shows an implementation in standard PostgreSQL.

The non-recursive part of the query implements exactly the initialization from lines 1-4 of the pseudocode in Figure 3.1. The recursive CTE therefore stores the distance value from the starting node as well as the last-hop node for each node in the graph. The distance value and the last-hop, hereafter collectively referred to as the node information of a node, are maintained or potentially updated per node over multiple iterations. It is important to note, that even if a node has not received an update within an iteration, exactly one entry with the current node information is kept for each node in the recursive CTE. Where n is the total number of nodes present in the graph, the recursive CTE eventually terminates after n iterations. It is then checked whether there are any negative cycles in the calculated paths. For this purpose, the cost values computed in the n-th iteration must not contain lower cost values for any nodes than the cost values estimated in iteration $n-1$, since a path from any node to any other node should contain at most $n-1$ edges. If a longer path with a lower cost is found anyway, that path must contain an advantageous negative cycle, so all result tuples are discarded and the function returns no output instead. Otherwise, the final distance values and last-hop nodes are read for each node. The actual path reconstruction per node is skipped to focus more on the target search.

The implementation in SQL has an extreme overhead compared to an implementation in a common programming language. The heavy load of the algorithm occurs in lines 6-13 of the pseudocode of Figure 3.1. Here, $n-1$ iterations are performed, each of which checks each edge

```sql
WITH RECURSIVE bellman_ford (cost, node, prev, iter) AS (
  SELECT :infinity, n.node, NULL :: int, (SELECT COUNT(*) FROM nodes)
  FROM   nodes AS n
  WHERE NOT node = :start
    UNION ALL
  SELECT 0, :start, NULL, (SELECT COUNT(*) FROM nodes)
    UNION ALL
 (SELECT DISTINCT ON (new.node) new.cost, new.node,
         new.prev, new.iter - 1
  FROM   bellman_ford AS dest, LATERAL (
    SELECT src.cost + e.cost, e.dest, e.src, dest.iter
    FROM   bellman_ford AS src, edges AS e
    WHERE  (e.src, e.dest) = (src.node, dest.node)
    AND    src.cost + e.cost < dest.cost
      UNION ALL
    SELECT dest.*
  ) AS new(wt, cost, node, prev, iter)
  WHERE  dest.iter > 0
  ORDER BY new.node, new.cost)
)
SELECT node, cost, prev FROM bellman_ford
WHERE  iter = 0 AND NOT EXISTS (
  SELECT 1
  FROM   bellman_ford AS last_iter, bellman_ford AS second_last_iter
  WHERE  last_iter.iter = 0 AND second_last_iter.iter = 1
  AND    last_iter.node = second_last_iter.node
  AND    last_iter.cost < second_last_iter.cost
);
```

Figure 3.2: Bellman-Ford realized in standard PostgreSQL

with an effort of $O(1)$. This is because a common programming language benefits from a context that can store arrays of distance values and last-hop nodes. These values for a given node can then be looked up by accessing the corresponding array fields. In the SQL implementation, we cannot expect to achieve similar efficiency, as we now have to consider the more inefficient join between tuples containing the information previously realized by arrays. Instead, we need to compute equi joins between three tables, those being the *edges* table and two instances of the recursive CTE containing the current node information. One instance of the latter is required to bind the node information for the source node u, another for the destination node v. There's even additional overhead required to copy the node information of nodes that weren't updated in the last iteration, and to ensure that for each node, only the node information with the lowest cost is kept for the next iteration. An optimized approach could therefore benefit greatly from the use of a hash table. First, this would avoid the need to copy tuples containing node information across iterations if certain nodes do not receive updates throughout the iteration, as the hash table tuples remain available between iterations if they are not updated or actively discarded. In addition, it is now possible to access the distance value of any node via an efficient

hash table lookup that benefits from index support. This means that we can completely avoid expensive joins between different tables, and instead estimate the required node information by just searching one bucket of tuples. An implementation of Bellman-Ford using the hash table is shown in Listing 3.1.

```sql
WITH RECURSIVE bellman_ford (wt, cost, node, prev) AS (
  SELECT -4, :infinity, n.node, NULL :: int
  FROM   nodes AS n
  WHERE NOT node = :start
    UNION ALL
  SELECT -4, 0, :start, NULL
    UNION ALL
  SELECT 0, (SELECT COUNT(*) FROM nodes), NULL, NULL
    UNION ALL
 (WITH node_info AS (SELECT * FROM bellman_ford AS ht),
       iter  AS (SELECT cost FROM bellman_ford AS wt0r),
       edges AS (SELECT e.*  FROM edges AS e, iter)
  SELECT res.*
  FROM   iter, LATERAL (
    SELECT 0, iter.cost - 1, NULL, NULL
      UNION ALL
    SELECT -4, cheapest_updates.*
    FROM (
      SELECT DISTINCT ON (new_dest) (hashtablelookup(2, e.src) :: int) +
             e.cost AS new_cost, e.dest AS new_dest, e.src
      FROM   edges AS e
      WHERE  hashtablelookup(2, e.src) + e.cost <
             hashtablelookup(2, e.dest)
      ORDER BY new_dest, new_cost
    ) AS cheapest_updates
  ) AS res
  WHERE  iter.cost > 1
    UNION ALL
  -- read out the hash table content into WT 1
  SELECT res.*
  FROM   iter, LATERAL (
    SELECT 1, cost, node, prev
    FROM   node_info
    WHERE NOT EXISTS (
      SELECT 1
      FROM   edges AS e
      WHERE  hashtablelookup(2, e.src) + e.cost <
             hashtablelookup(2, e.dest)
    )
  ) AS res
  WHERE  iter.cost = 1
 )
)
SELECT node, cost, prev FROM bellman_ford WHERE wt = 1;
```

Listing 3.1: Bellman-Ford using several WTs and a hash table

In this query, a *reset* WT is just used to store and update an iteration counter so that exactly n iterations are be performed. This allows a separation of the actual node information and the iteration counter, which were previously combined in a tuple of the main WT. Both the initialization and the updates of the node information are now done in the hash table, which is indicated by the upsert operations of the tuples with the *wt* column with value -4. Now that only relevant upsert statements are performed, node information that does not receive updates no longer needs to be copied. This requires the output of the function to be read from the hash table at the end, which is done by writing to a result WT 1. This allows the detection of negative cycles to be incorporated into the recursive CTE, i.e. writing to WT 1 only occurs in the absence of negative cycles. It is worth noting that the calculation of the *edges* CTE is mandatory. The reason for this is that SQL seems to assume that the output of a system information function is constant over the course of multiple calls. If the hash table lookup were instead to refer to node indexes read from the predefined *edges* table, the lookup values would be evaluated only once and then cached for future iterations. This corresponds to an SQL optimization, as the effort to compute the upsert tuples could apparently be endured only once rather than in every iteration. In reality, of course, this should not happen, because with updated cost values for the hash table entries, we also get updated lookup values. If this unwanted optimization were to occur, only the first iteration would compute upsert tuples that actually perform a state change. In future iterations, the same upsert tuples would be computed again, without any actual updates occurring and therefore no convergence to the optimal paths. The same problem occurs when calculating an *edges* CTE without any dependency on the *iter* CTE that contains the iteration counter. Since the *iter* CTE must be read in every iteration and cannot be computed just once over the course of the recursive CTE, the *edges* CTE must also be estimated in each iteration. This guarantees that with a new instance of the *edges* CTE, a new set of upsert tuples must be computed. This in turn ensures that the system information function calls are re-computed using the updated hash table instance, rather than just using cached information. Basically, we had to actively avoid an SQL optimization in order for the hash table lookup to work properly.

Now the first example and its optimization have been explained in detail. Another example is the Dijkstra algorithm, which has an almost identical goal and a somewhat similar implementation to the Bellman-Ford algorithm. However, there are other ways to optimize it. We will look at these in the next part.

3.1.2 Dijkstra

As with the Bellman-Ford algorithm, the Dijkstra algorithm [8] computes all paths from a starting node to all other nodes in the graph. However, while the Bellman-Ford algorithm generally tolerates edges with negative weights as long as there is no negative cycle in the computed paths, this is not the case for the Dijkstra algorithm. Because of this stronger constraint on the graph to only allow positive edge values, a more efficient algorithm can be developed where only a small subset of edges needs to be considered for updates within an iteration. The approach is that within an iteration, only one node and its outgoing edges are selected, i.e. only the directly reachable nodes are checked for cheaper paths. The next node to expand from should not

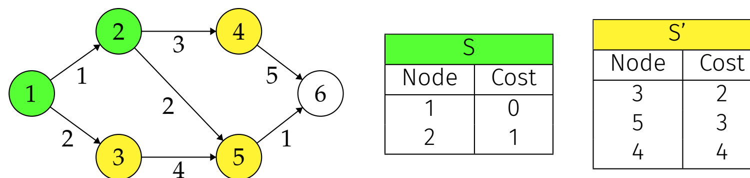


Figure 3.3: Example graph processed by the Dijkstra algorithm

have been expanded from before, and it should have the lowest cost value of all non-expanded nodes. This is the case because by not allowing negative edge weights at all, the cheapest path to the selected node must have already been computed. Thus it can be removed from the set of non-expanded nodes and expanded from to compute further updates for the reachable nodes. At the start, only the starting node is considered for an expansion, as its cost value obviously is 0. After expansion, the node information of the starting node is inserted into a set S containing node information for all previously expanded nodes. Again, for each node, the node information includes the node identifier, the cost to reach that node and the last-hop node identifier for a potential path reconstruction, which is not implemented again. After that, the iteration is completed and the next node to expand from is selected from the set of reachable but not yet expanded nodes. Let us call this set S'. Both in literature and in practice, it is advised to use a priority queue to store this set S', as this allows the node information to be ordered by cost value. This means that the node to be expanded from next with the lowest cost value of the set S' can easily be fetched without having to search through the entire contents of S'. Listing 3.3 shows an example graph for which some iterations of the Dijkstra algorithm have already been performed and the sets S and S' are already established.

The expansion started at the node with index 1. In the first iteration, the two nodes 2 and 3 received updated cost values and entries in the set S' were created. Since the node 2 is reachable at a lower cost than the node 3, it was expanded from next and therefore an entry is made in the set S for this node. Again, all nodes reachable from node 2 receive updated cost values if they are cheaper. By checking the set S' we can see that the node 3 with the lowest cost of 2 would be expanded in the next iteration. An optimized query could therefore benefit from the usefulness of the implemented heap data structure. Note, however, that unlike with the hash table, tuples within the heap data structure may not be updated. They may only be inserted via an enqueue and read as well as removed via a dequeue operation. This means that if a path was previously found to a particular node, and a cheaper path is found in a later iteration, a second entry for that node will be added to the heap. The consequence of this is that after performing a dequeue operation to find the next node to expand from, we must first ensure that this node has not already been expanded from due to a previous entry in the priority queue. Since the set S' is realized by the heap, the set S with the expanded nodes could be realized by a WT. This once again implements the assignment of different responsibilities to the different WTs and data structures. Hence, even without using the heap, it would at least be possible to implement the separation into different WTs. This can be further improved by choosing the update rule *extend* for the set S, so that the node information for already expanded

```sql
WITH RECURSIVE dij (wt, cost, node, prev) AS (
  SELECT -4, 0, :start, NULL :: int
    UNION ALL
  SELECT -3, e.cost, e.dest, :start
  FROM   edges AS e
  WHERE  e.src = :start
  AND NOT e.src = e.dest
    UNION ALL
 (WITH new AS MATERIALIZED (SELECT * FROM dij AS he
       WHERE hashtablelookup(2, node) = -1 LIMIT 1)
  SELECT res.*
  FROM   new, LATERAL (
    SELECT -4, new.cost, new.node, new.prev
      UNION ALL
    SELECT -3, new.cost + e.cost, e.dest, new.node
    FROM   edges AS e
    WHERE  e.src = new.node
    AND    hashtablelookup(2, e.dest) = -1
    AND NOT e.src = e.dest
  ) AS res)
) SELECT d.node, d.cost, d.prev FROM dij AS d WHERE wt = -4;
```

Figure 3.4: Dijkstra algorithm using both a heap and a hash table

nodes does not need to be copied over multiple iterations anymore. However, to make full use of the implemented data structures, the hash table could also replace the *extend* WT for the set S. In addition to preserving stored tuples over many iterations, the hash table has the advantage that we can now use efficient lookups to determine whether a particular node has already been expanded from. This is especially useful because, as explained above, any node fetched from the priority queue may actually have been expanded from earlier, in which case it should not be expanded from again. In addition, updates to a particular node information should only occur if the specified node has not yet been expanded from, as the most efficient node information is already present. Normally, these lookups to determine whether a node has already been expanded from earlier would require a full scan of the set S. Enabling a faster lookup is therefore useful, and an efficiency gain is to be expected. There's already more than one optimization possible for the Dijkstra algorithm, these are realized in Figure 3.4.

The set S is realized by using the hash table, while the heap implements the set S'. The benefit of the hash table comes especially from the two lookup calls. One of them is present in the estimate of the next *new* tuple that would be expanded from next. Even though only exactly one tuple is selected from the heap because of the *LIMIT 1* clause, it must necessarily fulfill the given condition. This means that several heap dequeue operations could still take place within an iteration until a tuple is found that satisfies the condition - that condition being the one that the node must not have been expanded yet. Similarly, nodes receiving updates to their node information must not have been expanded from either, marking the second occurrence of a hash table lookup. Once the query terminates, the final node information is fully stored in the

hash table. Since there is no need to implement a negative cycle detection, as it was the case with the Bellman-Ford algorithm, and since multiple upsert statements for a single node cannot occur, the upsert statements themselves can be interpreted as the output of the function.

Multiple optimizations were now shown in a single query. In order to better understand the individual optimizations, the later chapter on measurements 4 will consider each optimization both individually as well as in combination. This will also be the case for the A* algorithm, whose workflow is very similar to that of the Dijkstra algorithm. The A* algorithm is briefly explained in the next chapter.

3.1.3 A*

As demonstrated with the Dijkstra algorithm, the A* algorithm [9] also uses a separation into the sets S and S', where S once again contains the already expanded nodes and S' lists the reachable but non-expanded nodes. However, the A* algorithm aims to compute only the path from the starting node s to a single selected end node e, rather than the paths to all nodes as it was done in the two previous algorithms. For this purpose, an additional heuristic is given which provides an underestimated cost value for each node. This heuristic value of a node v determines the approximate cost of reaching the end node from v. In an example where the nodes represent cities and the edges represent roads, the actual cost to a destination city would depend on the streets taken. However, the heuristic as an underestimation could correspond to the distance to the destination city by air. Using the heuristic value for each node, the A* algorithm allows a more targeted search from the starting node to the end node, requiring fewer iterations than Dijkstra or Bellman-Ford. For the Dijkstra algorithm, the ordering criterion for the heap S' was the costs from the starting node s to the current node. Now the heuristic value for the remaining path to the end node e is also considered. The new ordering criterion for a node v is therefore the sum of the actual costs from s to v and the estimated costs from v to e. With this change, the new ordering criterion describes the estimated cheapest cost from s to e if a path through the node v is chosen. By always selecting the next node to be expanded according to this new ordering criterion, the path search can be implemented in a more directed way towards the end node. Once the end node itself would be expanded from, the cheapest path has been found, as the expansion of any other node would result in higher costs. This is due to the fact that the heuristic should always underestimate, which also implies that the heuristic value for the end node should be 0. Unlike the other two graph algorithms, the implemented query for the A* algorithm also includes a path reconstruction. In practice, the Distance-Vector-Routing algorithm that implements Dijkstra's functionality relies only on calculating the the optimal next-hop for routing packages across the internet. It is not always necessary to compute the complete paths and thus the path reconstruction can be ignored. For the A* algorithm, however, this reasoning cannot be followed. Also, a single path can be reconstructed without much more effort, which would have changed the running time of the Dijkstra and Bellman-Ford algorithms immensely. Let us look at the path reconstruction in more detail. Once the path search terminated and the optimal cost has been estimated, the set S contains all the nodes visited by the path. For each node, the last-hop node is stored in this

set, meaning that it is possible to build the path iteratively. This reconstruction part of the A* query can again benefit from the efficient lookups possible by using a hash table for the set S. This is shown in Listing 3.2, where the query omits the parts previously shown because of the strong resemblance of A* to Dijkstra.

```
WITH RECURSIVE astar (wt, total_cost_estimate, node, prev, cost_to_node,
    path) AS (
  -- non-recursive part
  ...
  -- recursive part
 (WITH reconstr AS (SELECT * FROM astar AS wt2r),
       finished AS (SELECT * FROM astar AS wt3r),
       new AS MATERIALIZED (
         SELECT * FROM astar AS he
         WHERE  NOT EXISTS (TABLE reconstr)
         AND    NOT EXISTS (TABLE finished)
         AND    hashtablelookup(2, node :: int) = -1 LIMIT 1)

  SELECT res.*
  FROM   new, LATERAL (
    -- we didn't expand from the goal node, prepare the next A* step
    ...
    WHERE NOT new.node = goal
      UNION ALL
    -- we expanded from the goal node, start path reconstruction
    SELECT 2, new.total_cost_estimate, new.node, new.prev,
           new.cost_to_node, ARRAY[new.node]
    WHERE  new.node = goal
  ) AS res
    UNION ALL

  -- path-reconstruction
  SELECT 2, NULL, NULL, hashtablelookup(4, prev) :: int, cost_to_node,
         prev || path
  FROM   reconstr
  WHERE  NOT hashtablelookup(4, prev) = -1
    UNION ALL
  SELECT 3, NULL, NULL, NULL, cost_to_node, path
  FROM   reconstr
  WHERE  hashtablelookup(4, prev) = -1
  )
) SELECT cost_to_node, path FROM astar WHERE wt = 3 LIMIT  1;
```

Listing 3.2: A* algorithm implementing path reconstruction using both a heap and a hash table

This query uses the TS to split the recursive CTE into different functions. The first part is the usual path search, as previously demonstrated for the Dijkstra problem. However, once the end node has been chosen for expansion, the cheapest path has already been found and the reconstruction part begins. This is instructed by computing a tuple for WT 2 for the first time. The *new* CTE, which is responsible for selecting the next node to expand from, will always be empty

from now on, since the additional condition on WT 2 being empty is no longer fulfilled. Instead, the *reconstr* CTE reading from WT 2 will be extended by the last-hop node in each iteration until the path was fully reconstructed. The result is then written to WT 3 so that it can be easily in the following parent query. Again, the reconstruction part can benefit from the hash table and its lookup.

Many of the new constructs can be implemented for the three graph algorithms, and in all cases runtime improvements are expected. Now that two of the three implemented data structures have been demonstrated, the next chapter will provide a use case for the remaining data structure - the stack.

3.2 Cursor Loop Application

Cursors [19] may be used in the more imperative PL/PGSQL. This programming language allows you to perform standard SQL queries, as well as imperative constructs such as the use of variables and loops. For SQL queries to be used within a PL/PGSQL procedure, code written in PL/PGSQL must constantly switch between the PL/PGSQL domain, which implements the imperative constructs, and the SQL domain, which executes the queries [7]. These context switches result in an overhead for using PL/PGSQL. However, the ability to write common imperative code seems far more attractive, as it is much easier for most users to understand than the relational approach of standard PostgreSQL. In particular, the *WITH RECURSIVE* construct is considered a more cumbersome means to an end, as complex queries are difficult to realize with recursive CTEs and the readability is worse than that of an iterative procedure expressed in PL/PGSQL. As mentioned in the motivation of this thesis, there is a research underway to automatically translate PL/PGSQL code into standard PostgreSQL code, meaning that a user may still express the easy to understand PL/PGSQL code while still benefiting from the better performance of standard PostgreSQL. However, PL/PGSQL's cursors are an obstacle to this, as it is possible to express procedures that perform terribly worse when realized in standard PostgreSQL. Let us look at this in more detail. A cursor may be defined for a particular target query, as shown in Listing 3.3.

```
1  CREATE FUNCTION read_word_orderings() RETURNS SETOF int AS $$
2  DECLARE
3    word_id INT4 := 0;
4    c CURSOR FOR (SELECT id FROM word_orderings ORDER BY ordr ASC);
5  BEGIN
6    OPEN c;
7    FETCH c INTO word_id;
8    WHILE found LOOP
9      RETURN NEXT word_id;
10     FETCH c INTO word_id;
11   END LOOP;
12   RETURN;
13 END
14 $$ LANGUAGE PLPGSQL;
```

Listing 3.3: Example procedure implementing a cursor loop written in PL/PGSQL

The cursor variable *c* is bound to the target query in the *DECLARE* section, while the target query is not evaluated yet. Only in the *BEGIN* section is the cursor opened, allowing it to fetch one row at a time from the target query. This is done in a *WHILE LOOP* statement until the target query is fully read. In general, it is also possible to perform other operations on a table row bound by the cursor variable, such as updating or deleting the row. Another use case of cursors is to return the cursor variable bound to a target query. This allows the caller of the procedure to handle the cursor on its own behalf, i.e. to fetch rows manually. However, returning a cursor is not subject of this chapter, as this cannot be implemented efficiently in standard PostgreSQL, not even with the additional features. So, let us focus on the sequential fetching of the target query content within a cursor loop, as shown earlier in Listing 3.3. To do this, let us consider the more elaborate example from Listing 3.4.

```
1  CREATE TABLE word_orderings (id int PRIMARY KEY,  ordr int);
2  CREATE TABLE word_contents  (id int PRIMARY KEY, word text);
3  CREATE TYPE result AS (content text, row_counter int);
4  CREATE FUNCTION pretty_print(threshold int) RETURNS SETOF result AS $$
5  DECLARE
6    row_counter INT4 := 0;
7    row_letter_count INT4 := 0;
8    row_content text := '';
9    word_id INT4 := 0;
10   word_letter_count INT4;
11   word_content text;
12   c CURSOR FOR (SELECT id FROM word_orderings ORDER BY ordr ASC);
13 BEGIN
14   OPEN c;
15   FETCH c INTO word_id;

16   WHILE found LOOP
17     SELECT wc.word, LENGTH(wc.word)
18     INTO   word_content, word_letter_count
19     FROM   word_contents AS wc
20     WHERE  wc.id = word_id;
21     IF row_letter_count + word_letter_count + 1 > threshold THEN
22       RETURN NEXT (row_content, row_counter) :: result;
23       row_counter := row_counter + 1;
24       row_letter_count := word_letter_count + 1;
25       row_content := word_content || ' ';
26     ELSE
27       row_content := row_content || word_content || ' ';
28       row_letter_count := row_letter_count + word_letter_count + 1;
29     END IF;
30     FETCH c INTO word_id;
31   END LOOP;

32   RETURN NEXT (row_content, row_counter) :: result;
33   RETURN;
34 END $$ LANGUAGE PLPGSQL;
```

Listing 3.4: *pretty_print* procedure using context switches between the PL/PGSQL and SQL domains, as marked in yellow

```sql
CREATE FUNCTION pretty_print(threshold int) RETURNS SETOF result AS
$$
WITH RECURSIVE t(wt, row_content, row_counter, row_letter_count,
    table_offset, done) AS (
  -- non-recursive part
  ...
  -- recursive part
 (SELECT res.*
  FROM   t AS wt0r, LATERAL (
    SELECT wo.id
    FROM   word_orderings AS wo
    ORDER BY wo.ordr
    OFFSET table_offset
    LIMIT 1
  ) AS new_word_id(id), LATERAL (
    ...
  ) AS res
  ...
  )
) ...;
$$ LANGUAGE SQL;
```

Figure 3.5: Inefficient implementation of the *pretty_print* function in pure SQL

The task of the procedure expressed by the *pretty_print* User Defined Function (UDF) is to concatenate given words into sentences according to a given order of the words. Any sentence or result row created must not exceed a given threshold for the number of letters in each row. Although it would be a terrible database design in practice, two separate tables *word_contents* and *word_orderings* are used to store the word information. So, to estimate the next word to be processed within an iteration, both tables must be operated on. This is done artificially so that a cursor variable c may be bound to a query that iterates over the word identifiers in an ordered fashion, while within an iteration a context switch to the SQL domain is required to read the actual word. If this query were to be automatically translated into a standard PostgreSQL variant, it would inherently require the *WITH RECURSIVE* construct to replace the loop. Each iteration of the loop expressed in PL/PGSQL would then correspond to an iteration in the recursive CTE realizing the procedure. However, as the binding to a row from the target query referred to by the cursor should only occur once per iteration, this becomes difficult to realize in a recursive CTE. We no longer have the option of simply fetching the next row returned by the target query, but have to loop over the query to find the next row. This is demonstrated in the recursive CTE in Figure 3.5, which realizes the PL/PGSQL procedure from above.

The less relevant parts of the query have been discarded as the focus is on the fetch of the next word identifier according to the ordering criterion. Access to the next row returned by the target query now occurs through additional *OFFSET n* and *LIMIT 1* clauses, i.e. exactly one row is fetched from the target query, the one that would have been returned next by the target query. It is quite obvious that computing a possibly large target query in each iteration just to fetch

```
CREATE FUNCTION pretty_print(threshold int) RETURNS SETOF result AS
$$
WITH RECURSIVE t(wt, row_content, row_counter, row_letter_count,
    word_id) AS (
  -- non-recursive part
  ...
  SELECT -1, NULL, NULL, NULL, id
  FROM (
    SELECT id
    FROM   word_orderings
    ORDER BY ordr DESC
  ) AS _(id)
    UNION ALL
  -- recursive part
 (WITH new_word_id AS (SELECT word_id AS id FROM t AS stp LIMIT 1),
       new_word    AS (
         SELECT word, LENGTH(word) AS letter_count
         FROM   word_contents AS wc, new_word_id AS nwi
         WHERE  wc.id = nwi.id
       )
  ...
  )
) ...;
$$ LANGUAGE SQL;
```

Figure 3.6: Efficient implementation of the *pretty_print* function in pure SQL using multiple WTs and a stack

a single row results in an immense overhead. However, with the new features in the modified version of PostgreSQL, this could be solved more efficiently. I will demonstrate this by using the stack data structure, as it allows rows to be deleted and has no restrictions on to its tuple scheme like the heap and the hash table. The problem from above, implemented with the stack, is shown in Figure 3.6.

Again, the less relevant parts of the query were omitted. In the first iteration of the recursive CTE, the target query is fully evaluated and the content is pushed to the stack. Since the procedural variant expressed a particular ordering criteria for the target query, it must now be reversed. This is because the stack is a Last-In-First-Out data structure, where the first row fetched from the stack corresponds to the last row pushed. A queue data structure would allow the same ordering criterion, as it would correspond to a First-In-First-Out data structure, i.e. the first tuple added to the data structure would also be removed first. However, the stack now allows one fetch from the stack in each iteration, meaning that it is finally possible to access the rows of the target query in an efficient way. While this is expected to improve the runtime of the procedure translated to standard PostgreSQL, and thus provides a viable alternative to using *OFFSET* and *LIMIT* clauses, it may not compete with the original procedure in PL/PGSQL in every regard. Usually, the advantage of using cursors for to loop over the results of a target query is that the target query, which may return a large result, does not need to be computed

prior to the loop. Instead, it can fetch one row at a time, eliminating the need to hold the potentially large result of the target query in memory. The advantage of cursor loops is therefore largely one of space efficiency. However, the variant realized with a recursive CTE and the stack requires the target query to be fully evaluated and pushed to the stack within the first iteration, meaning that we cannot benefit from the lower space requirement. Note also that not all of the rows computed by the target query may be relevant to the cursor loop, the loop may have an alternative termination condition and the computation of the remaining rows of the target query may be discarded. In this case, the realization using the stack in the recursive CTE would even have a runtime overhead, as it insists on fully evaluating the target query. It is evident that using the stack for cursor loop applications does not manage to improve on the procedural implementation in PL/PGSQL, but it still offers a better alternative to the current option of using *OFFSET* and *LIMIT* clauses to iterate over the target query.

The full potential of the stack was not used to realize cursor loops with pure SQL. In the next chapter, however, the stack will be used for its intended purpose.

3.3 CPS Examples

3.3.1 Introduction

Now the CPS examples, which were a major part of the motivation for this thesis, will be presented. These examples correspond to problems which conventionally are solved by recursive functions. As a representative for the CPS examples, the Floyd-Warshall algorithm [20] will be presented and modified in the following. This algorithm is another way of calculating the cheapest path cost in a directed and weighted graph, as explained in chapter 3.1. The pseudocode for the Floyd-Warshall algorithm is shown in Figure 3.7 [21].

Figure 3.7: Floyd-Warshall pseudocode

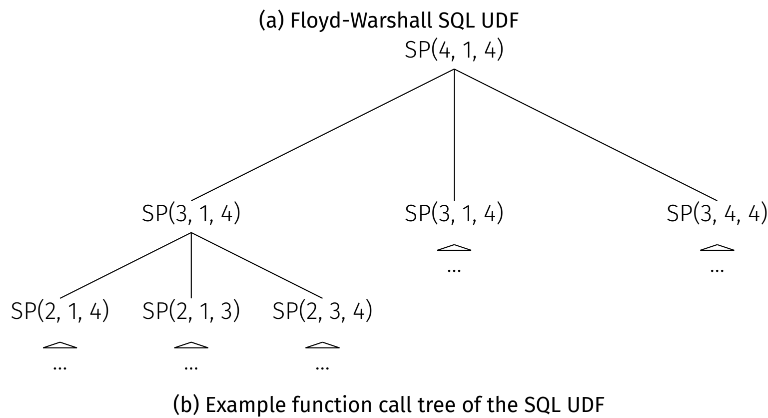
Algorithm 2 Floyd-Warshall

```
1: for k ← i to n do
2:     for i ← 1 to n do
3:         for j ← 1 to n do
4:             d[i,j] ← min(d[i,j], d[i,k] + d[k,j])
5:         end for
6:     end for
7: end for
```

The special feature of the Floyd-Warshall algorithm is that it benefits from dynamic programming, where the main problem is split into smaller sub-problems that are solved first and the results of which are stored for the computation of the main problem. In the pseudocode shown in Figure 3.7, this is achieved by using an iterative approach and an array d which is being filled with the intermediate results. An entry $d[i, j]$ stores the minimum cost of a path from node i to node j. Initially, only direct edges (u, v) from a node i to j are stored in the d array, while the d values for all node pairs that do not share an edge are initialized with ∞. Then the algorithm begins. In each iteration a node index variable k is incremented and for each pair of nodes i and j the d value is potentially updated. This is the case if a new path from i to j can be established

```sql
CREATE FUNCTION shortestpath(nodes int, s int, e int) RETURNS int AS
$$
  SELECT CASE
    WHEN nodes = 0 THEN (SELECT edge.weight
                         FROM   edges AS edge
                         WHERE  (edge.here,edge.there) = (s,e))
    ELSE (SELECT LEAST(shortestpath(nodes - 1, s, e),
                       shortestpath(nodes - 1, s, nodes) +
                       shortestpath(nodes - 1, nodes, e)))
  END;
$$ LANGUAGE SQL;
```

(a) Floyd-Warshall SQL UDF



(b) Example function call tree of the SQL UDF

Figure 3.8: Floyd-Warshall SQL UDF and function call tree

through node k that costs less than the previously stored path cost $d[i,j]$. This means that in each iteration, only the nodes $1,2,...,k$ may appear in the paths considered by the optimization algorithm. The functionality of the algorithm is based on the the following insight: Assume that the cheapest path costs have already been computed for all node pairs, but only paths through the nodes $1,2,...,k-1$ have been considered. Then the cheapest path cost for a pair of nodes (i,j) considering only nodes $1,2,...,k$ must either already be known, i.e. it does not go through the node k, or on the contrary it does, which means the cheapest path should be computed by $d[i,j] = d[i,k] + d[k,j]$. This is represented in line 4 of the pseudocode from Figure 3.7. The pseudocode represents a bottom-up approach to dynamic programming [22]. This bottom-up approach means that we start by calculating the smallest possible intermediate results and use these to construct the solutions to larger problems. In the following, we will instead consider a top-down approach known as memoization, where a larger problem is given and we then identify the sub-problems, which are then solved recursively. A top-down approach using a UDF is realized in Figure 3.8a.

For this top-down approach, a function call that computes the shortest path cost from a starting node s to an end node e in a graph with n nodes $1,2,...,nodes$ is issued via $shortestpath(nodes,s,e)$. The $nodes$ parameter corresponds to the k value used in the pseudocode

from Figure 3.7. A *shortestpath* call will therefore issue three *shortestpath* calls, each one using a decremented *nodes* value. This top-down handling of a function call is demonstrated in the function call tree from Figure 3.8b above, but no actual memoization may yet be used by its implementing pseudocode. This is because each result of a function call only occurs in the parent call, rather than them being stored in an array that can be accessed at any time. In addition to not being able to use memoization, the recursive UDF even suffers from a terrible complexity of $O(3^n)$ [5], which is a dramatic change from the usual complexity of $O(3^n)$. Fortunately, this can be fixed by using an adapted implementation with a recursive CTE. By translating the query into the CPS, [5] the previous recursive *shortestpath* calls are now replaced by tail calls. This means that after the intermediate result of a call has been computed, there is no outer context to return to in which the intermediate result would expect further computations. Instead, the function call is now extended with an additional argument, which is an anonymous function, the continuation. This continuation stores the outer context and it will be responsible for performing the remaining computations. The translation into the CPS using tail calls is necessary to turn the recursive problem into one that can be implemented using the *WITH RECURSIVE* construct. After that, the defunctionalization step is necessary, as SQL does not allow first-order-functions which are used by plain CPS. After this additional translation, the sequential workflow is now controlled by two separate functions, *Apply* and *Fun*. The workflow between the *Fun* and *Apply* functions is illustrated in Table 3.1.

Instruction call	Closure stack (cont, nodes, s, e, s1, s2)
Fun(4, 1, 4)	∅
Fun(3, 1, 4)	(1,4,1,4,0,0)
Fun(2, 1, 4)	(1,3,1,4,0,0), (1,4,1,4,0,0)
Fun(1, 1, 4)	(1,2,1,4,0,0), (1,3,1,4,0,0), (1,4,1,4,0,0)
Fun(0, 1, 4)	(1,1,1,4,0,0), (1,2,1,4,0,0), (1,3,1,4,0,0), (1,4,1,4,0,0)
Apply(∅)	(1,1,1,4,0,0), (1,2,1,4,0,0), (1,3,1,4,0,0), (1,4,1,4,0,0)
...	...
Fun(0, 1, 4)	(3,1,2,4,2,∅), (3,2,3,4,3,∅), (3,3,1,4,5,1), (1,4,1,4,0,0)
Apply(∅)	(3,1,2,4,2,∅), (3,2,3,4,3,∅), (3,3,1,4,5,1), (1,4,1,4,0,0)
Apply(2)	(3,2,3,4,3,∅), (3,3,1,4,5,1), (1,4,1,4,0,0)
Apply(3)	(3,3,1,4,5,1), (1,4,1,4,0,0)
Apply(4)	(1,4,1,4,0,0)
Fun(3, 1, 4)	(2,4,1,4,4,0)
Fun(2, 1, 4)	(1,3,1,4,0,0), (2,4,1,4,4,0)
Fun(1, 1, 4)	(1,2,1,4,0,0), (1,3,1,4,0,0), (2,4,1,4,4,0)
Fun(0, 1, 4)	(1,1,1,4,0,0), (1,2,1,4,0,0), (1,3,1,4,0,0), (2,4,1,4,4,0)
Apply(∅)	(1,1,1,4,0,0), (1,2,1,4,0,0), (1,3,1,4,0,0), (2,4,1,4,4,0)
...	...
Apply(∅)	(3,1,2,4,2,∅), (3,2,3,4,3,∅), (3,3,4,4,∅,∅), (3,4,1,4,4,4)
Apply(2)	(3,2,3,4,3,∅), (3,3,4,4,∅,∅), (3,4,1,4,4,4)
Apply(3)	(3,3,4,4,∅,∅), (3,4,1,4,4,4)
Apply(∅)	(3,4,1,4,4,4)
Apply(4)	∅
Finished(4)	∅

Table 3.1: Workflow of the Floyd-Warshall algorithm in CPS and TS

The *Fun* call now replaces the *shortestpath* calls and appends a closure record (k, env) to the next call. The closure record realizes the encoding of the continuation as data rather than as a function. For this purpose, it stores the environment of free variables given in the outer context and a value *clos* that selects the specific continuation. The outer context includes the arguments *nodes*, *s* and *e* as well as the intermediate results *s1* and *s2* to the sub-calls in lines 7 and 8 of the UDF from Figure 3.8a. The *Apply* call, unlike the *Fun* call, checks the most recent closure record and performs the functionality indicated by the continuation *cont*. This happens when a *Fun* call has produced an intermediate result, and now the remaining calculations on the previous outer context are to be initiated. Therefore, an *Apply* call receives the intermediate result computed by the *Fun* call as an argument and possibly checks the other intermediate results *s1* and *s2* given in the topmost closure. Note how the set of closures can easily be represented by a stack, since Apply calls usually read the last closure added. Furthermore, by adapting to the TS, the two functions *Fun* and *Apply* can be integrated into a single function responsible for controlling the workflow. The resulting function now just needs to call itself instead of calling either *Fun* or *Apply*. An introduced label must therefore differentiate between the two different function calls, so that the resulting function is able to decide whose functionality should be executed. With all of the above adjustments, an SQL function may be implemented that realizes a recursive CTE to solve the problem. Figure 3.9 shows such a query for the Floyd-Warshall algorithm.

In each iteration only one tuple is generated, this tuple corresponds to an instruction tuple. Its *label* column marks the functionality to be performed by the recursive CTE in the current iteration, indicated by either a *Fun* or an *Apply* label. A third label *Finish* denotes a result tuple that the function will eventually return as the solution to the top-down function call issued by the caller. Each instruction tuple is extended with a closure array, so that the outer contexts may be stored whenever a *Fun* call is issued. An additional *res* column has been added to the scheme of the recursive CTE to store an intermediate or final result. The form of the query differs from the form proposed in [6] in that it offers only two main blocks, one for *Fun* instructions to be read and one for *Apply* instructions. Since there is only one instruction per iteration, only one of the two SFW blocks is actually processed. This makes the query slightly more readable, but more importantly it allows for further improvements when using multiple WTs and memoization. The latter is not currently included in the query. Before the memoization will be realized, the stack shall first find a proper use, as promised in the last chapter. This will be demonstrated in the next chapter.

3.3.2 Closure Stack and multiple WTs

The basics of CPS queries were introduced in the previous chapter. A notable aspect of the implementation design was that the closures were now given by an array. This is inefficient because, first, we need to use the array functionality and, secondly and more importantly, the arrays need to be copied in each iteration. Since at most one closure may be removed from the array per iteration, and its size may grow very large over the course of a function call, this can lead to an expensive copy operation in each iteration. In the last chapter, it was already stated that the set of closures is best represented by a stack, since Apply instructions usually only

```sql
WITH RECURSIVE recurse(label, nodes, s, e, res, k) AS (
  SELECT 'Fun', :nodes, :s, :e, NULL :: int, ARRAY[] :: kontFloyd[]
    UNION ALL
 (WITH instr AS (SELECT * FROM recurse)
  -- Apply-call
  SELECT  _.*
  FROM    instr AS t, LATERAL (
    SELECT 'Finished', NULL, NULL, NULL, t.res, NULL
    WHERE  CARDINALITY(t.k) = 0
      UNION ALL
    SELECT 'Fun', t.k[1].nodes - 1, t.k[1].s, t.k[1].nodes, NULL,
           (2, t.k[1].nodes, t.k[1].s, t.k[1].e, t.res, 0)
           :: kontFloyd || t.k[2:]
    WHERE  t.k[1].cont = 1
      UNION ALL
    SELECT 'Fun', t.k[1].nodes - 1, t.k[1].nodes, t.k[1].e, NULL,
           (3, t.k[1].nodes, t.k[1].s, t.k[1].e, t.k[1].s1, t.res)
           :: kontFloyd || t.k[2:]
    WHERE  t.k[1].cont = 2
      UNION ALL
    SELECT 'Apply', NULL, NULL, NULL,
           LEAST(t.k[1].s1, t.k[1].s2 + t.res), t.k[2:]
    WHERE  t.k[1].cont = 3
  ) AS _
  WHERE   t.label = 'Apply'
    UNION ALL
  -- Fun-call
  SELECT  _.*
  FROM    instr AS t, LATERAL (
    SELECT 'Apply', NULL, NULL, NULL, (SELECT e.weight FROM edges AS e
           WHERE (e.here, e.there) = (t.s, t.e)), t.k
    WHERE  t.nodes = 0
      UNION ALL
    SELECT 'Fun', t.nodes-1, t.s, t.e, NULL,
           (1, t.nodes, t.s, t.e, 0, 0) :: kontFloyd || t.k
    WHERE  t.nodes <> 0
  ) AS _
  WHERE   t.label = 'Fun')
) SELECT t.res FROM recurse AS t WHERE t.label = 'Finished';
```

Figure 3.9: Floyd-Warshall algorithm in CPS and TS implemented via a recursive CTE

access the last closure added. In the following part the inefficient closure array will therefore be replaced by an actual closure stack. This avoids the need to copy the entire closure array in every iteration, as the stack content remains available until the termination of the recursive CTE unless actively removed. In addition, separate WTs will now be used for *Apply* and *Fun* instructions. The implementation of the Floyd-Warshall algorithm using a closure stack and multiple WTs is shown in Listing 3.5.

In this implementation, we can detect a clear separation of tuples according to their purpose. The WT 0 will now only contain *Apply* instructions, whereas the WT 1 will store *Fun* tuples. This means that the trampolization was included in the structure of the query instead of by using a *label* column. Each SFW block therefore only selects either WT 0 or 1 to read from. The *label* column has thus been integrated into the *wt* column, which means that the *label* column may be discarded. This saves us one column of memory per tuple. In addition, the stack instance is solely responsible for storing the closures. To do this, the closures must first be atomically encoded, i.e. each closure is now represented by its own tuple. Unfortunately, tuples performing a stack push are required to follow the scheme of the recursive CTE, meaning that the stack tuples will have the same attributes as the instruction tuples. This means that any attributes used by a tuple representing an instruction are irrelevant to a stack tuple, resulting in many columns filled with *NULL* values. Stack pushes that demonstrate this are marked yellow in the query from Listing 3.5. For a stack tuple, only the *wt* column and the *k* column for the actual closure content are relevant. Similarly, the instruction tuples given in WT 0 now obtain the value *NULL* instead of any closure content. A different scheme for stack tuples would be worthwhile to avoid the many *NULL* columns. In addition to the stack pushes performed, the top element of the closure stack is read from the stack in each iteration. This is indicated by the alias *st*, which means that the top element is simply read without being removed by an instant pop operation. This is necessary because the top element of the stack should only be be removed when reading an *Apply* instruction, not however when reading a *Fun* instruction. Due to this dynamic, pop operations must be actively instructed by computing a stack pop tuple indicated by a *wt* column -2. This is marked green in the query from Listing 3.5.

By using a closure stack, instruction tuples and closure contents may be clearly separated. Also, expensive copy operations to maintain the closure array over many iterations are now avoided. In the next chapter we want to avoid costly copy operations again, but this time by using an *extend* WT for memoization content instead of using a closure stack.

3.3.3 Memoization

As explained in the motivation for this thesis, the Floyd-Warshall algorithm and certain similar problems can benefit immensely from the use of memoization. Any intermediate result of a *Fun* call can therefore create a memoization entry containing both the arguments and the result of the call. This allows that if multiple *Fun* calls with the same arguments are issued throughout the recursive CTE, the result can simply be looked up. By looking up the stored result value, it is not necessary to perform the same calculation again. Whether a particular function may actually benefit from memoization depends on the function itself. This is only the case if multiple *Fun*

```sql
WITH RECURSIVE recurse(wt, nodes, s, e, res, k) AS (
  SELECT 1, :nodes, :s, :e, NULL :: int, NULL :: kontFloyd
    UNION ALL
 (WITH top   AS (SELECT * FROM recurse AS st LIMIT 1),
       clos  AS (SELECT clos.*
                 FROM (SELECT NULL) AS _ LEFT OUTER JOIN
                      (SELECT (t.k).* FROM top AS t)
                       ON true)
  -- Apply-call
  SELECT  _.*
  FROM    (SELECT * FROM recurse AS wt0r) AS t, clos, LATERAL (
    SELECT 2, NULL, NULL, NULL, t.res, NULL :: kontFloyd
    WHERE  clos IS NULL
      UNION ALL
    SELECT 1, clos.nodes - 1, clos.s, clos.nodes, NULL, NULL
    WHERE  clos.cont = 1
      UNION ALL
    SELECT -1, NULL, NULL, NULL, NULL,
           (2, clos.nodes, clos.s, clos.e, t.res, 0) :: kontFloyd
    WHERE  clos.cont = 1
      UNION ALL
    SELECT 1, clos.nodes - 1, clos.nodes, clos.e, NULL, NULL
    WHERE  clos.cont = 2
      UNION ALL
    SELECT -1, NULL, NULL, NULL, NULL,
           (3, clos.nodes, clos.s, clos.e, clos.s1, t.res) :: kontFloyd
    WHERE  clos.cont = 2
      UNION ALL
    SELECT 0, NULL, NULL, NULL, LEAST(clos.s1, clos.s2 + t.res), NULL
    WHERE  clos.cont = 3
      UNION ALL
    SELECT -2, NULL, NULL, NULL, NULL, NULL
  ) AS _
    UNION ALL
  -- Fun-call
  SELECT  _.*
  FROM    (SELECT * FROM recurse AS wt1r) AS t, LATERAL (
    SELECT 0, NULL, NULL, NULL, (SELECT e.weight FROM edges AS e
           WHERE (e.here, e.there) = (t.s, t.e)), NULL :: kontFloyd
    WHERE  t.nodes = 0
      UNION ALL
    SELECT 1, t.nodes-1, t.s, t.e, NULL, NULL
    WHERE  t.nodes <> 0
      UNION ALL
    SELECT -1, NULL, NULL, NULL, NULL,
           (1, t.nodes, t.s, t.e, 0, 0) :: kontFloyd
    WHERE  t.nodes <> 0
  ) AS _)
) SELECT t.res FROM recurse AS t WHERE t.wt = 2;
```

Listing 3.5: Floyd-Warshall algorithm in CPS and TS using multiple WTs and a closure stack

calls with the same arguments occur during the function call, i.e. during the recursive CTE. For the Floyd Warshall example, this can be easily comprehended by looking at the previously explained function call tree in Figure 3.8b, where during a function call *Fun(4, 1, 4)*, multiple occurrences of the *Fun* call *Fun(3, 1, 4)* are generated. The duplicate *Fun* calls would normally have to be executed in their entirety and independently of each other. Considering that this applies to the entire *Fun* call and all of its possibly many sub-calls, i.e. the entire sub-tree of the call *Fun(3, 1, 4)* shown in Figure 3.8b, it becomes clear that the runtime is needlessly increased. So, using memoization to store the arguments and the result of a completed *Fun* call will be advantageous. A variant of the Floyd-Warshall query that now implements memoization is shown in Listing 3.6.

The color yellow in Listing 3.6 marks the computation of a memoization tuple. These tuples use the parameter attributes *nodes, s* and *e* and the result attribute *res* to store an argument-to-result mapping. In order for the tuples to be preserved across all of the iterations of the recursive CTE, they must be copied in each iteration, marked green in Listing 3.6. To help with this, the CTE *memo_all* is created, which is used to separate the instruction tuples from the memoization tuples. The actual lookup for an existing memoization entry for the given parameters is only done in the block reading *Fun* instructions. If a memoization tuple was actually present for the given parameters of the *Fun* call, this memoization content is used in an additional SFW block, indicated by the color red in Listing 3.6.

While this approach alone yields a much better performance on its own, it can be further improved by using multiple WTs. As previously demonstrated for the closure stack in Listing 3.5, the WT 0 may again only contain *Apply* instruction tuples, while the WT 1 contains *Fun* instructions. Therefore the workflow is controlled by the tuples in WTs 0 and 1. Since the WT 2 is used for the final output tuple of the top-down function call, the next available WT 3 could be made responsible for storing the memoization contents. This avoids the need to separate the instruction and memoization tuples via CTEs. By assigning the update rule *extend* to the memoization WT 3, we can even avoid the copy statement of the entire *memo_all* CTE given in the SFW blocks marked green in Listing 3.6. This small and simple adjustment should further improve the performance in a meaningful way. In addition to not having to copy the memoization contents, also the *memo_all* CTE no longer has to be computed in every iteration. This is due to the automatic SQL optimization previously stated, where a CTE with only one reference in the following parent query is inlined to where it is used. While this optimization was actively disabled by using the *AS MATERIALIZED* specification whenever defining a CTE that implements a pop operation on a data structure, this now becomes useful. Since memoization lookups are only performed when reading a *Fun* instruction, any iteration reading an *Apply* tuple may completely ignore the computation of the *memo_all* CTE, further reducing the runtime of the query. Obviously, an *extend* WT for memoization purposes as well as a closure stack may both be implemented in a query, so that we may profit from both optimizations at once. This is also taken into account in the later measurements chapter. In principle, memoization could even be improved by using the index support provided by the hash table. The advantage of the hash table over an *extend* WT is quite obvious. Both the *extend* WT and the hash table are able to store tuples throughout the recursive CTE, leading to the optimization just explained. However, by using a hash table it is possible to perform the memoization lookup faster. Normally, the whole set of memoization

```sql
WITH RECURSIVE recurse(label, nodes, s, e, res, k) AS (
  -- non-recursive part
  ...
  -- recursive part
 (WITH recurse  AS (SELECT * FROM recurse),
       instr    AS (SELECT * FROM recurse WHERE label IS NOT NULL),
       memo_all AS (SELECT * FROM recurse WHERE label IS NULL)
  -- Apply-call
  SELECT  _.*
  FROM    instr AS t, LATERAL (SELECT (t.k[1]).*) AS clos, LATERAL (
    ...
    SELECT NULL, ((t.k[2]).args).nodes, ((t.k[2]).args).s,
    ((t.k[2]).args).e, LEAST(clos.s1, clos.s2 + t.res), NULL
    WHERE  clos.cont = 3
      UNION ALL
    SELECT * FROM memo_all
    WHERE  clos.cont <> 4
  ) AS _
  WHERE   t.label = 'Apply'
    UNION ALL
  -- Fun-call
  SELECT  _.*
  FROM    instr AS t, LATERAL (SELECT (t.k[1]).*) AS clos, LATERAL (
    SELECT COALESCE(m.avail, false) AS avail, m.val
    FROM (SELECT NULL) AS _ LEFT OUTER JOIN
         (SELECT true AS avail, m.res AS val
          FROM   memo_all AS m
          WHERE  (t.nodes, t.s, t.e) = (m.nodes, m.s, m.e)) AS m
          ON true
    ) AS memo, LATERAL (
    SELECT 'Apply', NULL :: int, NULL :: int, NULL :: int, memo.val, t.k
    WHERE  memo.avail
      UNION ALL
    ...
    SELECT NULL, (clos.args).nodes, (clos.args).s, (clos.args).e,
    (SELECT e.weight FROM edges AS e
     WHERE (e.here, e.there) = (t.s, t.e)), NULL
    WHERE  NOT memo.avail AND t.nodes = 0
      UNION ALL
    SELECT * FROM memo_all
  ) AS _
  WHERE   t.label = 'SP')
) SELECT t.res FROM recurse AS t WHERE t.label = 'Finish';
```

Listing 3.6: Floyd-Warshall algorithm in CPS and TS using memoization

tuples would have to be searched for an existing memoization entry that matches the given parameters. With index support, the lookup again only needs to consider a small subset of tuples. Also, the *memo_all* CTE no longer needs to be calculated. While the hash table can generally be considered a more powerful tool than the *extend* WT, its implementation also comes with an obstacle: The hash table lookup can only be performed by using a single integer key. This is an issue for the CPS examples, as most of them use multiple parameters for a *Fun* call, and thus the memoization lookup. This means that in the current implementation, the hash table may not be used for most of the CPS examples. Only the Fibonacci [23] and Faculty problems can benefit from the implemented hash table, as both problems require only one integer parameter. However, the Faculty itself cannot benefit from memoization because no *Fun* call with certain arguments occurs more than once during a function call. The Fibonacci problem, on the other hand, does have multiple occurrences of the same *Fun* calls, which is why this example will be considered. Figure 3.10 shows a query for the computation of the n-th Fibonacci number using a memoization hash table.

Memoization contents are now inserted into the hash table instead of an *extend* WT. This allows a faster hash table lookup of the result $fib(n)$ stored in the second column of a memoization tuple by providing the parameter n as the key. As the results of a Fibonacci call become very large very quickly and the integer range is easily exceeded, numeric values with a much wider range of values were allowed as result values in addition to pure integers. Again, it is important to note that the lookup value of -1 correspond to an unsuccessful lookup where the key was not found in the hash table. For this reason, the *WHERE* clauses always compare the estimated lookup value with -1 to determine whether the value is actually legitimate.

In this chapter, memoization was enhanced by both an *extend* WT and a hash table, so that the memoization contents do not have to be copied in every iteration. Note that only a limited kind of memoization was considered, where the memoization tuples of the WT only remain available until the termination of the recursive CTE, i.e. until the end of a specific function call. Only *Fun* calls within the same function call may thus benefit from the memoization contents, as other function calls use different and independent instances of the recursive CTE. The memoization contents computed throughout a single function call could in principle be inserted into a defined *memo* table, then even multiple function calls could benefit from the memoization contents. However, this will be disregarded for the measurements. These are carried out and evaluated in the next chapter.

```
WITH RECURSIVE recurse(wt, res, x, k) AS (
  SELECT 1, NULL :: numeric, :n, array[ROW(3, NULL, :n) :: kontFib]
    UNION ALL
 (-- Apply-call
  SELECT  _.*
  FROM    (SELECT * FROM recurse AS wt0r) AS t, LATERAL
          (SELECT (t.k[1]).*) AS clos, LATERAL (
    SELECT 2, t.res, NULL :: int, NULL :: kontFib[]
    WHERE  clos.cont = 3
      UNION ALL
    SELECT 1, NULL, (clos.num - 2 ) :: int,
           ROW(2, t.res, clos.num - 2) :: kontFib || t.k[2:]
    WHERE  clos.cont = 1
      UNION ALL
    SELECT 0, t.res + clos.num, NULL, t.k[2:]
    WHERE  clos.cont = 2
      UNION ALL
    SELECT -4, t.res + clos.num, (t.k[2]).args :: int, NULL
    WHERE  clos.cont = 2
  ) AS _
    UNION ALL
  -- Fun-call
  SELECT  _.*
  FROM    (SELECT * FROM recurse AS wt1r) AS t, LATERAL
          (SELECT hashtablelookup(2, t.x) AS val) AS memo, LATERAL (
    SELECT 0, memo.val, NULL :: int, t.k
    WHERE  memo.val > -1
      UNION ALL
    SELECT 0, 1, NULL, t.k
    WHERE  memo.val = -1 AND (t.x = 1 OR t.x = 2)
      UNION ALL
    SELECT -4, 1, t.x, NULL
    WHERE  memo.val = -1 AND (t.x = 1 OR t.x = 2)
      UNION ALL
    SELECT 1, NULL, t.x - 1, ROW(1, t.x, t.x-1) :: kontFib || t.k
    WHERE  memo.val = -1 AND t.x > 2
  ) AS _)
) SELECT t.res FROM recurse AS t WHERE t.wt = 2;
```

Figure 3.10: Fibonacci algorithm in CPS and TS using multiple WTs and a hash table for memoization purposes

4 Measurements

Finally, the runtime improvements are measured. The measurements were taken on a 64-bit Linux machine containing two AMD EPYC™7402 CPUs at 2.8 GHz and 2 TB of RAM. The PostgreSQL version is 14.1 and only the modified version of PostgreSQL presented in the implementation chapter will be considered for the measurements. To represent standard PostgreSQL, each example provides one query labelled *Standard* using only one *reset* WT 0. This query does not use any new constructs and could therefore be evaluated in standard PostgreSQL, so it will be used as the baseline query for comparison with all modified queries. Each individual query is evaluated five times and the average runtime in *ms* will is displayed either in a tabular way or as a graph. For each query using the hash table, 100 buckets are used. This ensures that hash table accesses do not correspond to simple array accesses, as usually far more than 100 elements are added to the hash table. The results of the measurements are presented in the same order as they were previously introduced in chapter 3. So, the next chapter shows the results for the graph algorithms.

4.1 Graph Algorithms

The graph algorithms measured include Bellman-Ford, Dijkstra and A*. For all of the graph algorithms, the runtime is calculated with respect to the graph size, i.e. the number of nodes present in the graph. Cycles are generally possible for the generated graphs, but negative edge weights have been avoided completely by randomly choosing the weight values from the interval $[1, 50]$. The probability of an edge between two nodes depends on the graph size. It has been adjusted so that on average, each node has five outgoing edges. We start with the algorithm that has the least number of different queries to measure, which is Bellman-Ford. The node with index 1 is always chosen as the starting node, from which the expansion starts. Two of the relevant queries were previously shown in detail in the chapter 3.1.1. The *Standard* query as usual only implements one *reset* WT and thus has to copy or update the path information of each node within each iteration, as previously shown in chapter 3.1.1 in Figure 3.2. This results in a lot of copy operations, but more importantly, an expensive join between three tables. In contrast, the measured *HT* query avoids copy operations by storing the node information in the hash table. However, it does not use the index support provided by the hash table to perform efficient lookups. This is only done by the query *HT Lookup*, which was previously shown in Figure 3.1 from chapter 3.1.1. The results of the measurements are shown in the graph in Figure 4.1. The results are quite interesting, especially since the *HT* query does not benefit the runtime. Let us check how many copy operations are actually prevented for a graph with 300 nodes. The

Standard query computes a total of 90 300 tuples. This adds up, because 300 tuples are required for initialization, and over the course of 300 iterations, 300 tuples are generated or copied in each iteration. In contrast, only 1590 tuples are computed by the recursive CTE of the *HT* query, which corresponds to only about 1.76% of the tuples required by the *Standard* query. It is astonishing that by avoiding almost 90 000 copy operations, no benefit in runtime can be achieved. So, the bottleneck in the computation cannot be in the copy operations. Let us have a closer look at the *HT* query. When returning every single tuple generated by the recursive CTE, instead of just the result tuples, it becomes evident that after 13 iterations no more upsert tuples are generated. The optimal paths are therefore found after only 13 iterations, because then no more upsert tuples are computed, i.e. the computed optimal paths are fairly short. However, the computation to convergence after 13 iterations does not correspond to a large part of the runtime of the recursive CTE. This can be proven by again returning every single tuple instead of the actual output of the function, and using a *LIMIT* clause to restrict the number of result tuples. It is the remaining 287 iterations that need to be performed regardless, which hurt the performance a lot. It becomes clear that the huge overhead is the expensive join between the *edges* table and the two instances of the *node_info* CTEs, which are computed in each iteration. This huge overhead is prevented by the *HT Lookup* query, which uses hash table lookups to avoid the expensive joins. Its runtime improvement over the other two queries is immense, as it now only needs to iterate over the *edges* table during an iteration, instead of computing a join over three tables. So, for the Bellman-Ford algorithm, the only real benefit of the hash table comes with the faster lookup using the index support. An argument can certainly be made that once the graphs reach a huge size, even the omitted copy operations will matter eventually. 90 000 tuples do not seem to matter yet, but we will have a closer look at this later. As a final note, the *HT* query has a strange behaviour where its runtime seems to vary a lot. As mentioned above, a value shown in the graph represents the average of five runs of a query. While any query can have some outliers in terms of the runtime, this is especially crucial for the *HT* query. For example when using 300 nodes, the *HT* query would sometimes have a runtime of about 17 000 ms, and other times only about 10 000 ms. This is a drastic change in runtime, which seems to occur due to different plans being used. Whenever the more efficient plan is used multiple times, the average runtime is also greatly reduced, which can be seen for 240 nodes and the resulting kink in the graph. Obviously, the planner in SQL cannot yet take advantage of the hash table and use its underlying structure to construct or decide for better plans. However, this could and probably should be taken into account, when adding the hash table to a productive environment. Next, the Dijkstra measurements are shown in Figure 4.2.

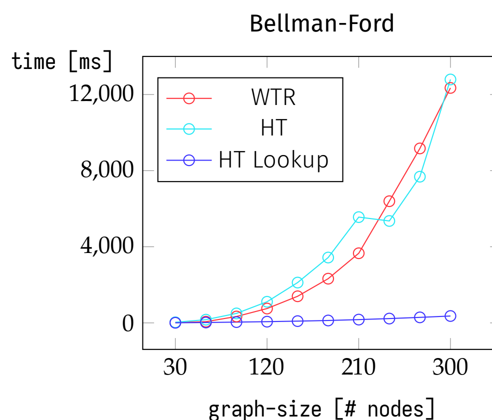


Figure 4.1: Bellman-Ford measurements

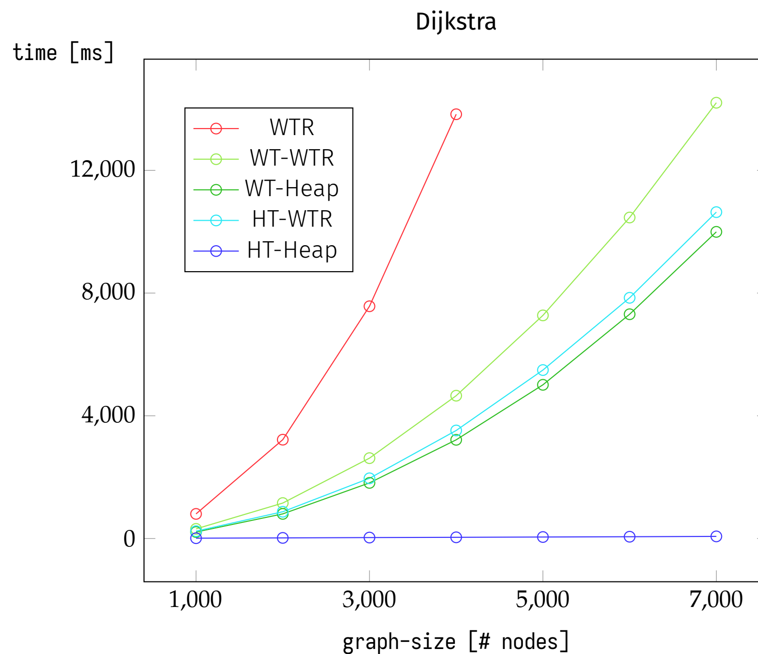


Figure 4.2: Dijkstra measurements

Again, the node with index 1 was chosen as the starting node for the measurements. Note that the graph sizes are chosen much larger than for the Bellman-Ford algorithm. However, the queries can still process them quite fast because only one node is expanded in each iteration instead of all of them. As expected, the *Standard* query is the least performing one. Using a graph with 4000 nodes, the *Standard* query computes a total of 7 890 378 tuples for the set S containing the already expanded nodes. Since some of the nodes are unreachable from the starting node, only 3972 tuples are computed for the set S in the case of the *WT-WTR* query, which uses an *extend* WT for the set S and a *reset* WT for the set S'. So, unlike the Bellman-Ford queries, where there was no real benefit to be gained from preventing copy operations by using a hash table, the sheer number of copy operations avoided for the Dijkstra algorithm has a noticeable impact on runtime. All other queries measured for the Dijkstra algorithm use either an *extend* WT or a hash table to store the tuples of the set S, i.e. they all avoid the copy operations. Therefore, the *WT-WTR* query could be considered the baseline for the remaining optimizations. Now, there seem to be two bottlenecks for the runtime. One of these is the inefficiency of having to copy or update the set S', which is currently handled by a *reset* WT. The other is to check whether a particular node has already been expanded from in a previous iteration, this check occurs several times per iteration. The *HT-WTR* query, which now uses a hash table instead of an *extend* WT to store the set S, solves the latter bottleneck. Using the indexing support to check if a particular node has been expanded before is useful and actually provides a runtime benefit. The *WT-Heap* query, on the other hand, sticks to a simple *extend* WT for the set S, but implements a heap for the set S'. While the *WT-WTR* query computes a total of 4 862 104

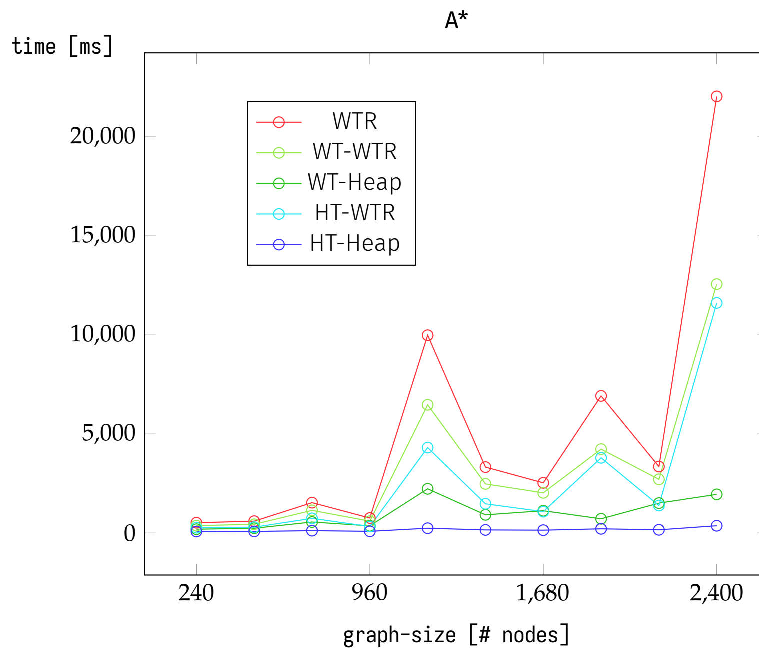


Figure 4.3: A* measurements

tuples for the set S', the *WT-Heap* query only requires 9989 tuples. As with the implementation of an *extend* WT for the set S, the bottleneck is again prevented by avoiding copying tuples that are still needed in future iterations. The optimizations reach a truly respectable point when both a hash table for the set S and a heap for the set S' are used. This is implemented by the query *HT-Heap*. Because it solves both of the bottlenecks mentioned above, the runtime drops to a low level that allows much larger graphs to be handled with ease. To put this into perspective, for a graph with 4000 nodes, this query generates only a total of 13 961 tuples, while other queries require many millions of tuples or implement some other form of efficiency throughout the recursive CTE. The same improvements have also been implemented for the A* algorithm. Its measurements are shown in the graph from Figure 4.3.

For all queries, 100 paths are computed from random starting nodes to the end node with index 10. An adapted Dijkstra query was used to compute the heuristic values required by the A* algorithm. To do this, each edge weight is underestimated by up to 30%. The spikes in the runtime across all of the different query variants are explained due to longer paths being computed. This results in more iterations and therefore a higher chance of selecting an irrelevant node for expansion when trying to expand towards the end node. Unlike the Dijkstra queries, the A* queries benefit greatly from the targeted search towards the end node. Therefore, both the *WT-WTR* and *HT-WTR* queries, each of which uses either an *extend* WT or a hash table to store the set S, have less of an impact on the runtime than they did for the Dijkstra queries. A much smaller number of nodes are required to be expanded to find the optimal path, so the set S does not grow as large anymore, and saving copy operations is therefore not as much

of a bottleneck. However, the heap data structure for the representation of the set S' is still important, especially now that the handling of the set S' is the main bottleneck. Using 1200 nodes for the graph, the *WT-WTR* query computes on average 45 317 tuples for the set S' per function call, while the *WT-Heap* query requires only 713 tuples. Over the course of 100 function calls, this adds up to a lot of copy operations that are prevented by the using the heap. Not surprisingly, the *HT-Heap* query is again the best performer, as it implements all the optimizations at once. Therefore, all three graph algorithms could benefit greatly from the implemented constructs, making an implementation of each of these algorithms feasible for an SQL environment. Following these measurements, the cursor loops application is be measured in the next chapter.

4.2 Cursor Loop Application

For the measurement of the *pretty_print* functions introduced in the chapter 3.2, the threshold 100 is always chosen. The queries are evaluated once as demonstrated, i.e. two separate tables *word_contents* and *word_orderings* are used to enforce context switches for the PL/PGSQL procedure between the SQL and PL/PGSQL domains. The variants using a plain SQL implementation must therefore compute joins between the tables. In order for this to remain feasible, primary keys have been added to the *id* columns of the tables. In addition, the queries have also been implemented in such a way that they only require a single table *words*, and therefore the PL/PGSQL procedure does not require any context switches, and the recursive CTEs do not need to perform any joins between the tables. Although the runtimes for the queries with and without context switching are not fully comparable, a lot of information can still be gathered from this. The measurements are shown in Figure 4.4.

Query	10 000 Words	1 000 000 Words
Naive	26 236	-
PL/PGSQL	18	1485
Stack	71	6031
Naive CS	11 359	-
PL/PGSQL CS	65	6111
Stack CS	83	7383

Figure 4.4: Cursor loop runtime measurements in *ms*

The queries marked with *CS* use context switches or joins, i.e. multiple tables are used. The others, on the other hand, use only one table. Since the naive query using a recursive CTE with *OFFSET* and *LIMIT* clauses has a terrible runtime, each query had to be evaluated with a rather small number of words, 10 000. The measured values speak for themselves, the naive implementation is just ridiculously bad. This demonstrates that it's just not feasible to translate certain cursor loops into standard SQL at the moment. To compare the PL/PGSQL procedure with the new variant using a recursive CTE and a stack, measurements were made using 1 000 000 words. Again, the PL/PGSQL procedures end up giving the better performance. Note, however, that the PL/PGSQL procedure and the query using a recursive CTE and a stack are much closer in their runtime when multiple tables and thus context switches are enforced. The PL/PGSQL procedure

does suffer a lot from context switches, but not so much that it performs worse than the stack implementation. It is save to say that a translation from PL/PGSQL code to plain SQL code would now be possible and somewhat feasible, if the stack data structure were added to recursive CTEs. However, we cannot ignore that the that the translation of PL/PGSQL code using cursor loops would not improve the runtime, even if context switches are given for the procedure. Perhaps in the future the stack data structure could be optimized to further reduce the stack query runtime, for example by allowing a stack scheme independent of the scheme of the recursive CTE, or even by allowing atomic stack values instead of tuples, so that stack accesses are faster. But even then, one cannot ignore that a recursive CTE, which computes an instruction tuple per iteration to update the state usually given in the PL/PGSQL variables, comes with its own overhead. Each individual variable accessed by the cursor loop must be specified as an attribute of the recursive CTE scheme, which inflates the scheme. Also, not every variable is updated in every iteration, so the attribute would just have to be copied unnecessarily. So, even if the stack implementation were given in an SQL environment, it would be questionable whether cursor loops should be translated into SQL code for efficiency reasons.

4.3 CPS examples

Finally, the eight selected examples [5] already available in the CPS and the TS are evaluated. Half of them may benefit from using memoization, as during a function call multiple *Fun* calls are generated with the same arguments. For these examples, a query that implements memoization but still uses only a *reset* WT is chosen as the baseline *Standard Memo* query to compare with the optimized queries. Also, a *Standard* query is always provided so that it is possible to see how much of an effect memoization itself has. As only a *reset* WT is used, the memoization contents must still be copied in each iteration. This is not the case for the query *Memo-WT*, which uses separate WTs for the *Apply* and *Fun* instructions, thus replacing the the former *label* column, and an additional *extend* WT for storing the memoization tuples. Copy operations of memoization contents are therefore avoided by this query. Both the baseline query *Standard* as well as the optimized query *Memo-WT* are extended by the use of a closure stack, the resulting queries are *Stack* and *Memo-WT & Stack*. For the queries that do not benefit from memorisation, only the *Standard* and *Stack* queries are implemented, i.e. only the efficiency gain for replacing the closure array by a closure stack is measured. Let us first start with the measurements for the Floyd-Warshall algorithm described in chapter 3.3 and the Dynamic-Time-Warping algorithm, which also benefits immensely from memoization. For the Floyd-Warshall algorithm, 10 function calls are issued for the paths from the starting node with index 1 to the end nodes with indexes $1-10$. The number of nodes in the graph is chosen in the interval $[9,20]$ and will be visible as the x-axis in the graph. The Dynamic-Time-Warping algorithm uses sequences with sizes of 100, and the *iterations* parameter, which represents the x-axis of the graph, determines how many consecutive function calls with incremented arguments are issued. The graphs with the measurements for these two examples are shown in Figure 4.5.

Both of the examples show a similar behaviour, so the following evaluation applies to both.

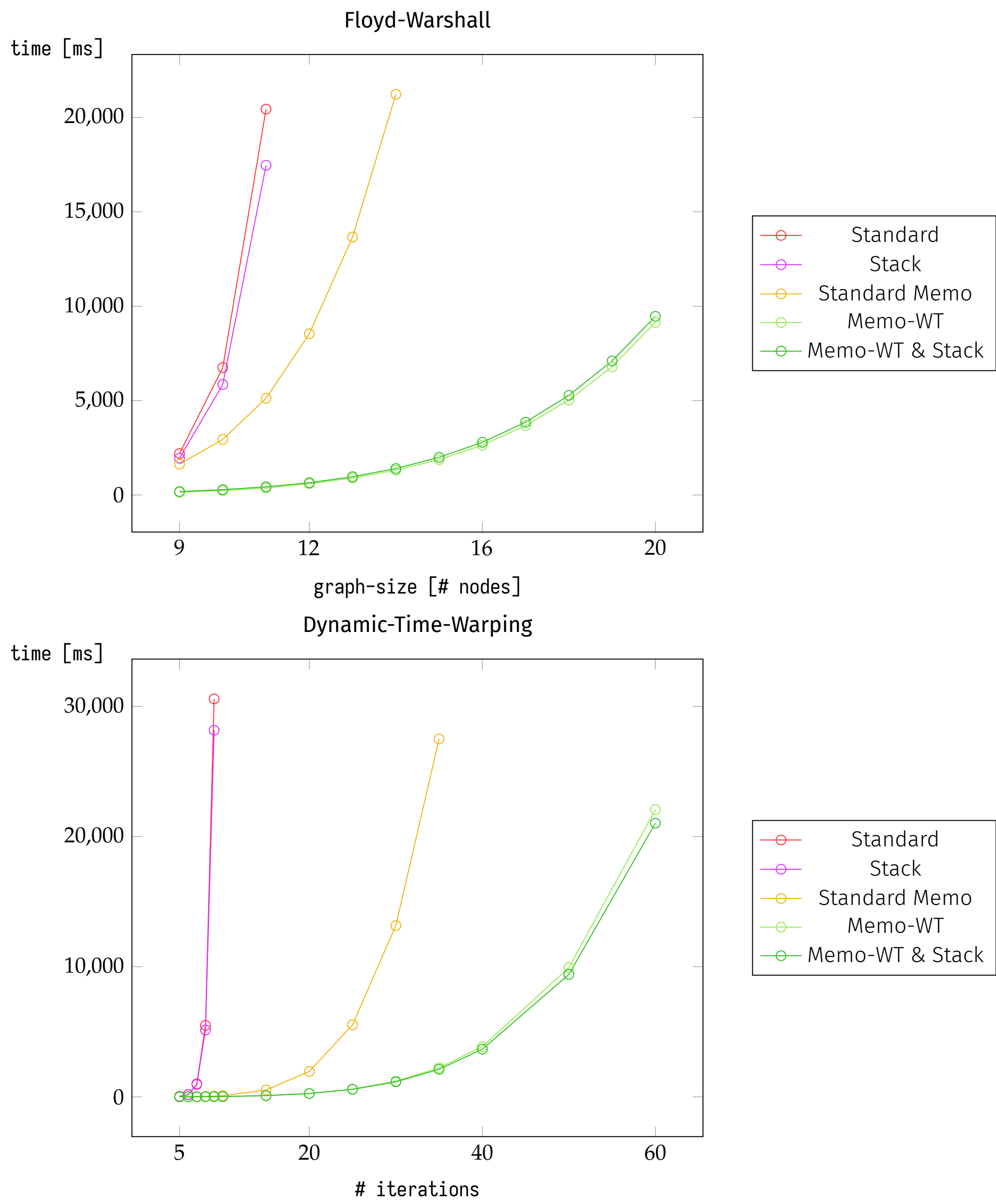


Figure 4.5: Measurements for Floyd-Warshall and Dynamic-Time-Warping

First, note how the use of memoization in the *Standard Memo* query visibly reduces the runtime of the *Standard* query, and how this allows much larger arguments to be used. In addition, the *Memo-WT* query shows a significant performance gain over the *Standard Memo* query just by not having to copy the memoization tuples in each iteration. For a graph with 13 nodes, the *Standard Memo* query for the Floyd-Warshall algorithm computes an average of 1 857 047 memoization tuples over the course of the recursive CTE. The *Memo-WT* query, on the other hand, only requires 857 memoization tuples to be computed. Again, a huge runtime optimization has been achieved by avoiding the copying of useful tuples. Also, as explained in the chapter 3.3.3, the CTE *memo_all* only needs to be computed in an iteration where a *Fun* instruction is read, because the computation of the CTE will be inlined into the *Fun* SFW block due to only occurrence of the *memo_all* CTE in this SFW block. This further reduces the evaluation effort of the *Memo-WT* query. Now let us take a look at the use of the closure stack. For the Dynamic Time-Warping algorithm, using the stack always results in a slight improvement in runtime. For the Floyd-Warshall algorithm, this is only the case if memoization is not used. When using 11 nodes for the Floyd-Warshall algorithm, the *Standard* query has an average closure array size of 10.5, while the *Standard Memo* query averages at 9.6. Note that the top-level function call of the *Standard Memo* query already requires one stack push to enable memoization, i.e. the closure size of the *Standard* query would be even larger if it implemented the missing stack push as well. Different closure array sizes make sense because if there is a memoization tuple that matches the parameters of a *Fun* call, there is no need to descend into more *Fun* calls and therefore no additional stack pushes are performed. By using memoization, there are also far fewer iterations overall. The *Standard* query requires on average 531 440 iterations for a graph with 11 nodes. So, it is obvious that copying an array with even a small average size of 10.5 elements adds up when it has to be copied half a million times. Using the stack data structure is therefore a great help in avoiding these copy operations. In contrast to that, the *Standard Memo* query takes on average 2304 iterations to complete, i.e. the savings from using the stack simply don't justify its use. Note that the stack does have an overhead, as each closure entry must now correspond to an own tuple, which must be modified by special stack operations. The low performance of the closure stack shown is contrasted with the Fibonacci example, for which the corresponding measurements are given in the graph from Figure 4.6.

The x-axis shows the argument x for a Fibonacci function call. The *Standard* and *Stack* queries have been discarded for the graph because their runtimes are already around $10\ 000ms$ for $x = 30$. Memoization is the way to go here. As shown for Floyd-Warshall and Dynamic-Time-Warping, the Fibonacci queries also benefit immensely from not having to copy memoization tuples. But let us focus on the use of the stack and the hash table, starting with the former. For the input $x = 4000$, the average closure stack size is 2000 over the course of 15 996 instructions. This is a much larger stack size than for the examples previously demonstrated. It becomes clear that as the stack size increases, so does the positive impact on runtime, as the number of prevented copy operations skyrockets. Especially when a large stack has to be copied over many iterations, as is the case with the Fibonacci problem, the copying effort becomes considerable. One more optimization using the hash table instead of an *extend* WT for memoization purposes is measured. This allows fast and easy lookups for a possibly existing memoization tuple for the

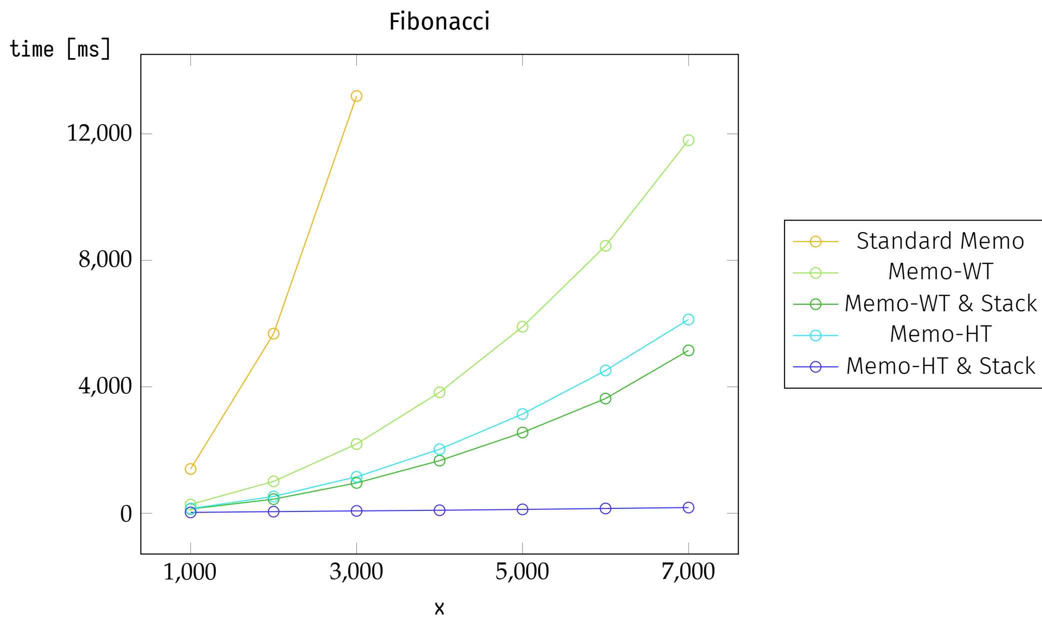


Figure 4.6: Fibonacci measurements

given arguments of a *Fun* call. In addition, the *memo_all* CTE previously shown in 3.3.3 no longer needs to be computed for the implementation of lookups. As the number of memoization tuples computed increases, so does the need for the hash table, as iterating over the entire memoization content for a single lookup otherwise becomes cumbersome. Overall, it is safe to say that using a hash table instead of a *extend* WT is always more profitable, as it serves the same purpose, but also offers additional benefits. Finally, the query *Memo-HT & Stack* is obviously the best performer. It prevents unnecessary copy operations by using both a closure stack to preserve closure contents and a hash table to store memoization tuples consistently. Fast lookups are also possible, so there is no bottleneck for this query. For completeness, the results for the remaining CPS examples [5] are shown in the tables from Figure 4.7.

Of the five remaining CPS examples, only the *lcs* example, which refers to the calculation of the longest-common-subsequence, can benefit from memoization. 100 different sequences with lengths between 7 and 8 are generated. 400 random pairs of sequences are evaluated in one function call each to measure the *Standard* and *Stack* queries, while 1500 pairs are used to test the different memoization queries. Again, a behaviour comparable to the Floyd-Warshall algorithm can be detected, i.e. the memoization itself is much improved when using an *extend* WT, and the stack loses its relevance because of reduced closure stack sizes and fewer iterations due to memoization. All the other examples cannot benefit from memoization and therefore only implement the *Standard* and *Stack* queries. The *comps* example computes the connected components in a directed acyclic graph. To generate the graph, the parameter n was set to 10, i.e. up to 2^n nodes were inserted. A total of 3000 function calls were issued, where each function call computes whether two random nodes are connected. The *eval* example corresponds to an

Query	comps	eval	fac	march
Standard	4032	2275	4001	7850
Stack	4593	2067	501	3032

(a) Measurements of the examples *comps, eval, fac, march*

Query (lcs)	400 calls	1500 calls
Standard	4976	-
Stack	4624	-
Standard Memo	1644	6152
Memo-WT	432	1612
Memo-WT & Stack	602	2261

(b) Measurements of the *lcs* example

Figure 4.7: Measurements of the remaining CPS examples

interpreter for arithmetic expressions. The parameter n, which determines both the number of generated expressions, again up to 2^n, and their potential depth, was set to 15. A total of $15\,000$ function calls and therefore expression evaluations were issued. For the Faculty *fac*, a single function call was issued with the argument $x = 15\,000$. Finally, the *march* example implementing the Marching Squares algorithm used 10 invocations and 200 iterations. With these selected parameters, only the Faculty and the Marching Squares algorithms, in addition to the Fibonacci example shown earlier, were able to benefit significantly from the use of a closure stack. While all the other optimizations ended up having a positive impact on the performance, the closure stack fell short of what was expected with only three out of eight examples taking advantage of it. There are some features that can be improved, however, and these will be discussed in the final chapter that follows.

5 Conclusion and Future Work

The goal of this thesis was to demonstrate modifications to standard PostgreSQL that would improve upon the *WITH RECURSIVE* construct. Now several tuple containers are available for use instead of just one. These range from simple WTs to more specific data structures such as the tuple heap. The separation of the tuples computed by the recursive CTE into different tuple containers allows the assignment of individual responsibilities to these tuple containers. As a consequence, the TS can naturally be integrated into the structure of the recursive CTE by assigning different instruction tuples to their own WTs. By declaring a WT with the *extend* update rule instead of the *reset* rule, the latter being common for the single instance of a WT in standard PostgreSQL, tuples may even be retained throughout the course of the recursive CTE, preventing expensive copy operations if the tuples remain relevant for more than one iteration. The same is true for all implemented data structures, they only differ in how the tuples are stored and how certain tuples can be read from the data structure. The tuple stack implements a Last-In-First-Out data structure, where the last element pushed is removed first. While a normal queue implements a First-In-First-Out data structure, the tuple heap, as a priority queue, inserts the tuples in an ordered fashion and thus allows easy fetching of tuples with minimal ordering criterion. The hash table has the advantage of index support, which means that tuples indexed by a particular key can be read, deleted and overwritten in an efficient way. All of these additional data structures provide more flexibility and efficiency in handling certain tuple data than standard WTs do. The measurements of a selected number of examples sometimes showed an immense decrease in their runtime, which supports the assumption that the additional features should lead to huge improvements of the *WITH RECURSIVE* construct. Many of the runtime optimizations came from not having to copy tuples that would be needed in future iterations, but would be discarded after one iteration of the recursive CTE. This was expected, as mentioned in the introduction to this thesis, because it is still required to copy all state variables over each iteration for a recursive CTE to solve a particular problem, since only immutable tables and WT tuples may be accessed in the body of the recursive CTE. The graph algorithms introduced were able to benefit from using the heap and the hash table to represent nodes and their node information. Cursor loops may now use a stack for them to be implemented in pure SQL instead of PL/PGSQL. While this doesn't give a better runtime than its PL/PGSQL counterpart, and it doesn't help with saving memory like cursors usually do, it at least makes an SQL implementation feasible. A select number of recursive UDFs that were already translated into CPS and TS could be improved in some cases by a closure stack. More importantly, examples that profit from memoization could benefit greatly from not having to copy many memoization tuples in each iteration. An index support on the tuple container that stores the memoization contents is also advantageous. Still, some features could see some improvements, these are

```
WITH RECURSIVE <instance-definition> [, <instance-definition>]* AS (
  // non-recursive part
  <insertion-statement>,  [<insertion-statement> ,]*

  // recursive part
  WITH <with-declaration> [, <with-declaration>]*
  <insertion-statement> [, <insertion-statement>]*
) <SFW-block>;

<instance-definition> :=
  <alias> (<scheme>)                          // for WTs
  <alias> {STACK | HEAP} (<scheme>)           // for Stacks or Heaps
  <alias> HT(buckets: x, keys: y) (<scheme>)  // for Hash-Tables

<with-declaration> :=
  <cte-alias> AS (SELECT ... FROM <alias> WHERE <cond> LIMIT <n>)

<insertion-statement> :=
  INSERT INTO <instance-name>
  <SFW-block> [UNION [ALL] <SFW-block>]*

<ht-deletion-statement> :=
  DELETE FROM <ht-name> WHERE <cond>
  DELETE FROM <ht-name> ON (<key>)
```

Figure 5.1: Syntax suggestions for the new constructs

going to be presented in the following outlook for possible future work. First, a proposal for the syntax to be used for an actual implementation of the features will be presented.

As explained in chapter 2.4, the implementation of the features for this modified version of PostgreSQL relies on hacks that extend the available syntax constructs with additional semantics. This results in the need to adapt to the given syntax constructs and thus in certain limitations for the implementation. Many of these could be removed if a proper syntax for the new features were implemented. A suggested syntax for the new recursive CTE is shown in Figure 5.1.

The first thing to note is that the new instance definitions following the *WITH RECURSIVE* clause now allow multiple tuple containers to be defined. Each instance definition should specify a name for the tuple container and optionally which type of container it represents, the options being *STACK*, *HEAP* and *HT* for a hash table. If no type is specified, a common WT is assumed. The *extend* WT was discarded, as the hash table seems to be more useful in any case. To define a hash table instance, the user could provide additional *bucket*, *keys* and *upsert* attributes. The *bucket* parameter would specify how many buckets are allocated for the hash table, and the *keys* parameter would determine the number n of key values required for a hash table lookup. The first n attributes of the scheme would then be selected as the concatenated key while the remaining attributes would become the obtainable value. This means that for the first time it may be possible to return a record instead of an atomic value. The next attribute

of the hash table scheme with index $n+1$ could then be interpreted as the upsert value, i.e. existing hash table entries would be updated if they had more optimal values in that column. An unsuccessful hash table lookup could also return a *NULL* tuple. Active delete statements for the hash table should still be possible. An association to be deleted could either be selected directly via an *ON* clause, which provides the key of the association, or the deletion could be performed depending on the outcome of a usual *WHERE* clause. For the heap, the first attribute could always be chosen as the ordering criterion, or an identical approach to the hash table could be followed. In addition, it would now be possible to use multiple instances per data structure, which was previously only the case for WTs. Multiple instances of a particular data structure could theoretically be useful for certain queries. For example, the Dijkstra and A* examples could in principle also use two hash tables if the heap data structure were not available, one each for the sets S and S'. As done with multiple WTs, an instance of a particular data structure should only be created when instructed to do so by an instance definition.

Now let us look at the scheme of the tuple containers. First, since the tuple containers are now defined in the instance definitions behind the *WITH RECURSIVE* clause, the extra *wt* column used throughout the thesis would no longer be needed. It would therefore be necessary for the parent query following the recursive CTE to have access to all the individual tuple containers. Typically, the result of the recursive CTE could be viewed as a tuple set filtered for results by the parent query. However, with a missing *wt* column, tuples from different tuple containers could no longer be separated in the parent query, meaning that individual accesses to the different tuple containers would be mandatory. This is further enforced by allowing different schemes for the individual instances. This would allow even two different WT definitions to use a separate scheme for their respective tuples. This again is reinforced by the questionable implementation of the closure stack, where each atomic closure was required to match the scheme of the instruction tuples, and vice versa, even though this would result in many column values with *NULL*. However, by using different schemes for the individual tuple containers, the tuples to be inserted into different tuple containers may no longer be computed by multiple SFW blocks connected by *UNION ALL* clauses, as the schemes of the generated tuples would not match. Therefore, the insertion of tuples into a particular tuple container should now be separated from the insertions into other tuple containers by a delimiter such as a comma, as shown in Figure 5.1 above. However, an insertion statement for a particular tuple container can still use multiple *UNION ALL* or even *UNION* connected SFW blocks. Each insertion statement would now name the target tuple container by an additional *INSERT INTO* clause, which is followed by the SFW blocks computing the tuples to be inserted. No major syntax changes are proposed for reading the data structures. Using CTEs to access the data structures within the recursive part seems to be a decent approach to reading exactly what is required, especially when also using *WHERE* and *LIMIT* clauses. It also guarantees that the order of operations performed on the data structure is correct, as read and delete operations are always performed first. To ensure that this is really the case, both reads from the stack and the heap should always count as pop operations. Also, inlining of their CTEs should be prohibited by always automatically selecting the *AS MATERIALIZED* variant for these CTEs. The hash table lookups could also work simply by specifying the key in a *WHERE* clause. In summary, the proposed changes would take the imple-

mented features one step further and allow them to be used in a slightly more convenient and efficient manner. However, they do not in any way simplify the recursive CTE, which is already a somewhat difficult to use. Adding special syntactic features to allow additional constructs will inevitably make things more complex. The appendix gives a query for the Floyd-Warshall algorithm, which uses this proposed syntax to implement both a closure stack and memoization.

This idea could be expanded by allowing an even more diverse set of data structures. As shown for the cursor loop application, a simple queue could be useful instead of having to use the stack. Admittedly, adding these features to a proper PostgreSQL installation would be cumbersome. In this thesis only an implementation of the constructs was realized and example queries were modified. Although syntax suggestions are also given, their implementation would not be trivial. In addition, other aspects such as the optimization of the planner need to be considered. It needs to be aware of the implemented constructs, how they behave and how they may be exploited to build more consistent and more efficient query plans. As we have seen, there are still plenty of opportunities to build on what has been researched. Even though the new features probably will not see themselves in an official PostgreSQL release anytime soon - or ever - the much better performance should certainly motivate the optimization of the recursive CTE.

6 Appendix

```
1  WITH RECURSIVE apply_call (res),
2                 fun_call   (nodes, s, e),
3                 closure_st STACK   (cont, nodes, s, e, s1, s2, args),
4                 memo_ht    HT(buckets: 100, keys: 3)   (nodes, s, e, res),
5                 result     (val) AS (
6    -- non-recursive part
7    INSERT INTO fun_call
8    SELECT :nodes, :s, :e
9    ,
10   INSERT INTO closure_st
11   SELECT 4, NULL, NULL, NULL, NULL, NULL, (:nodes, :s, :e) :: args
12   ,
13   -- recursive part
14  (WITH fst  AS (SELECT * FROM closure_st LIMIT 1),
15        snd  AS (SELECT * FROM closure_st LIMIT 1),
16        memo AS (SELECT h.res AS val FROM fun_call AS f, memo_ht AS h
                WHERE (h.nodes, h.s, h.e) = (f.nodes, f.s, f.e))
17   INSERT INTO result
18   SELECT a.res -- case 3
19   FROM   apply_call AS a, fst
20   WHERE  fst.cont = 4
21   ,
22   INSERT INTO fun_call
23   SELECT fst.nodes - 1, fst.s, fst.nodes -- case 4
24   FROM   apply_call, fst
25   WHERE  fst.cont = 1
26     UNION ALL
27   SELECT fst.nodes - 1, fst.nodes, fst.e -- case 5
28   FROM   apply_call, fst
29   WHERE  fst.cont = 2
30     UNION ALL
31   SELECT f.nodes - 1, f.s, f.e -- case 2
32   FROM   fun_call AS f, memo
33   WHERE  memo IS NULL AND f.nodes <> 0
34   ,
35   INSERT INTO apply_call
36   SELECT LEAST(fst.s1, fst.s2 + a.res) -- case 6
37   FROM   apply_call AS a, fst
38   WHERE  fst.cont = 3
39     UNION ALL
40   SELECT memo.val -- case 0
41   FROM   fun_call AS f, memo
42   WHERE  memo IS NOT NULL
```

```
43      UNION ALL
44    SELECT (SELECT e.weight FROM edges AS e WHERE (e.here, e.there) = (f.s,
         f.e)) -- case 1
45    FROM   fun_call AS f, fst, memo
46    WHERE  memo IS NULL AND f.nodes = 0
47    ,
48    INSERT INTO closure_st
49    SELECT 2, fst.nodes, fst.s, fst.e, a.res, 0, (fst.nodes - 1, fst.s, fst
        .nodes) :: args -- case 4
50    FROM   apply_call AS a, fst
51    WHERE  fst.cont = 1
52      UNION ALL
53    SELECT 3, fst.nodes, fst.s, fst.e, fst.s1, a.res, (fst.nodes - 1, fst.
        nodes, fst.e) :: args -- case 5
54    FROM   apply_call AS a, fst
55    WHERE  fst.cont = 2
56      UNION ALL
57    SELECT new.prio, new.cont, new.nodes, new.s, new.e, new.s1, new.s2, new
        .args
58    FROM   fun_call AS f, LATERAL (
59      SELECT 2 AS prio, (1, f.nodes, f.s, f.e, 0, 0, (f.nodes - 1, f.s, f.e
          ) :: args) -- case 2
60      WHERE  memo IS NULL AND f.nodes <> 0
61        UNION ALL
62      SELECT 1, fst.* FROM fst -- cases 0-2
63      ORDER BY prio
64    ) AS new(prio, cont, nodes, s, e, s1, s2, args)
65    ,
66    INSERT INTO memo_ht
67    SELECT (snd.args).nodes, (snd.args).s, (snd.args).e, LEAST(fst.s1, fst.
        s2 + a.res) -- case 6
68    FROM   apply_call AS a, fst, snd
69    WHERE  fst.cont = 3
70      UNION ALL
71    SELECT (fst.args).nodes, (fst.args).s, (fst.args).e, (SELECT e.weight
        FROM edges AS e WHERE (e.here, e.there) = (f.s, f.e)) -- case 1
72    FROM   fun_call AS f, fst, memo
73    WHERE  memo IS NULL AND t.nodes = 0
74    )
75  ) SELECT val FROM result;
```

Listing 6.1: Floyd-Warshall algorithm implementing memoization and a closure stack using the proposed syntax

Bibliography

[1] PostgreSQL. URL: https://www.postgresql.org/.

[2] D.A. Varvel and L. Shapiro. “The computational completeness of extended database query languages”. In: *IEEE Transactions on Software Engineering* 15.5 (1989), pp. 632–638. DOI: 10.1109/32.24712.

[3] Implementation of recursive CTEs in PostgreSQL. URL: https://doxygen.postgresql.org/nodeRecursiveunion_8c_source.html.

[4] Recursive CTE. URL: https://www.postgresql.org/docs/current/queries-with.html.

[5] Tobias Burghardt, Denis Hirn, and Torsten Grust. “Functional Programming on Top of SQL Engines”. In: *Practical Aspects of Declarative Languages*. Ed. by James Cheney and Simona Perri. Cham: Springer International Publishing, 2022, pp. 59–78. ISBN: 978-3-030-94479-7.

[6] Denis Hirn and Torsten Grust. “A Fix for the Fixation on Fixpoints”. In: (2023). URL: https://db.cs.uni-tuebingen.de/publications/2023/a-fix-for-the-fixation-on-fixpoints/.

[7] Denis Hirn and Torsten Grust. “One WITH RECURSIVE is Worth Many GOTOs”. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 723–735. ISBN: 9781450383431. DOI: 10.1145/3448016.3457272. URL: https://doi.org/10.1145/3448016.3457272.

[8] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. 2022, pp. 287–290.

[9] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[10] WorkTableScanState in PostgreSQL. URL: https://doxygen.postgresql.org/structWorkTableScanState.html.

[11] CTE Materialization in PostgreSQL. URL: https://www.postgresql.org/docs/current/queries-with.html#id-1.5.6.12.7.

[12] PostgreSQL: Using Explain. URL: https://www.postgresql.org/docs/current/using-explain.html.

[13] Robert Rönngren and Rassul Ayani. “A Comparative Study of Parallel and Sequential Priority Queue Algorithms”. In: *ACM Trans. Model. Comput. Simul.* 7.2 (Apr. 1997), pp. 157–209. ISSN: 1049-3301. DOI: 10.1145/249204.249205. URL: https://doi.org/10.1145/249204.249205.

[14] Index Types in PostgreSQL. URL: https://www.postgresql.org/docs/current/indexes-types.html.

[15] Ward Douglas Maurer and Ted G Lewis. “Hash table methods”. In: *ACM Computing Surveys (CSUR)* 7.1 (1975), pp. 5–19.

[16] PostgreSQL System Information Functions. URL: https://www.postgresql.org/docs/9.4/functions-info.html.

[17] Richard Bellman. “On a routing problem”. In: *Quarterly of applied mathematics* 16.1 (1958), pp. 87–90.

[18] Michael J. Bannister and David Eppstein. “Randomized Speedup of the Bellman–Ford Algorithm”. In: *2012 Proceedings of the Meeting on Analytic Algorithmics and Combinatorics (ANALCO)*, pp. 41–47. DOI: 10.1137/1.9781611973020.6. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973020.6. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611973020.6.

[19] PostgreSQL Cursors. URL: https://www.postgresql.org/docs/current/plpgsql-cursors.html.

[20] Robert W. Floyd. “Algorithm 97: Shortest Path”. In: *Commun. ACM* 5.6 (June 1962), p. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168. URL: https://doi.org/10.1145/367766.368168.

[21] Chandler Burfield. “Floyd-warshall algorithm”. In: *Massachusetts Institute of Technology* (2013). URL: https://pdfs.semanticscholar.org/c053/e94fb1bc48c7354dcd3e8d78dc41a44d2768.pdf.

[22] David B Wagner. “Dynamic programming”. In: *The Mathematica Journal* 5.4 (1995), pp. 42–51. URL: http://yaroslavvb.com/papers/wagner-dynamic.pdf.

[23] Dan Kalman and Robert Mena. “The Fibonacci Numbers—Exposed”. In: *Mathematics Magazine* 76.3 (2003), pp. 167–181. DOI: 10.1080/0025570X.2003.11953176. eprint: https://doi.org/10.1080/0025570X.2003.11953176. URL: https://doi.org/10.1080/0025570X.2003.11953176.