EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-
Naturwissenschaftliche
Fakultät

Datenbanksysteme

Masterthesis

Using Postgres Hash Table Extension in Compiled Functional-Style SQL UDFs

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme

Madeleine Mauz

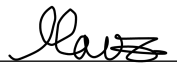
From 1st November, 2021 to 30th April, 2022

Examiner: Prof. Dr. Torsten Grust, Universität Tübingen
Co-Examiner: Prof. Dr. Klaus Ostermann, Universität Tübingen
Supervisor: Christian Duta, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit 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 Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, 10. April 2022



Madeleine Mauz

Acknowledgements

I would like to express my gratitude to my thesis supervisor Christian Duta for his invaluable advice, continuous support and guidance during my thesis. I could always ask whenever I ran into a trouble spot or had a question.

I wish to extend my special thanks to Denis Hirn for providing the Postgres hash table extension.

Finally, I would like to thank my parents and my partner for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Abstract

The size of Common Table Expressions (CTEs) in PostgreSQL 13.4 can grow arbitrarily and with it the runtime of the CTE scan. However there is no option for a CTE index to increase the performance of joins and lookups. This thesis proposes the usage of a hash table extension to simulate a CTE index, which increases the performance immensely especially for large CTEs. Two different types of hash table application were considered. In the first static approach, the data is written to a hash table and then only used for reading. The second dynamic approach allows reading and writing data on the hash table in arbitrary order. The method was applied on functional-style UDF templates using several different recursive algorithms, where varying performance improvements could be observed. Using the hash tables inside UDFs is a manual process, which requires special care to ensure a specific execution order, that is necessary for a correct query execution.

Zusammenfassung

Die Größe von Common Table Expressions (CTEs) in PostgreSQL 13.4 kann beliebig wachsen und dadurch auch die Laufzeit des CTE-Scans. Es gibt jedoch keine Möglichkeit für einen CTE-Index, um die Geschwindigkeit von Joins und Lookups zu erhöhen. In dieser Arbeit wird die Verwendung einer Hashtabellenerweiterung vorgeschlagen, um einen CTE-Index zu simulieren, der die Leistung insbesondere bei großen CTEs immens steigert. Es wurden zwei verschiedene Arten der Anwendung von Hashtabellen betrachtet. Beim ersten statischen Ansatz werden die Daten in eine Hashtabelle geschrieben und dann nur noch zum Lesen verwendet. Der zweite dynamische Ansatz erlaubt das Lesen und Schreiben von Daten in die Hashtabelle in beliebiger Reihenfolge. Die Methode wurde auf UDF-Templates im funktionalen Stil mit verschiedenen rekursiven Algorithmen angewandt, wobei unterschiedliche Leistungsverbesserungen beobachtet werden konnten. Die Verwendung von Hashtabellen innerhalb von UDFs ist ein manueller Prozess, der besondere Sorgfalt erfordert, um eine bestimmte Ausführungsreihenfolge zu gewährleisten, die für eine korrekte Funktionsausführung notwendig ist.

Contents

1

Introduction

Common Table Expressions in PostgreSQL 13.4 are temporary tables that exist just for one query. [9] Although the CTE size can grow arbitrarily and with it the runtime of the CTE scan, there is no option to create an index for a CTE. Indexes can increase database performance immensely if they are used correctly. [2] A temporary index on a CTE could significantly improve query performance.

In this work, we create temporary indexes on CTEs using hash tables as a proof of concept. It is implemented by manually filling a hash table with rows of the CTE and use selected columns as key to query rows.
In addition to the static index, where the hash table is completely filled with the CTE rows and then used only for reading, we implement a more dynamic index, which is continuously updated. Rows are already read before all rows are written. Both methods of using hash tables in CTEs are able to increase the performance of the query.

Large User Defined Functions (UDFs) with many CTEs are introduced in the paper *Functional-Style SQL UDFs With a Capital 'F'* by Duta and Grust. [12] We apply the proposed methods to some of these UDFs to enhance their performance.
The UDFs that use hash tables gain a big performance boost, which can be seen in figure 1.1. The dynamic time warping functions generated with the template from the paper and a manually optimized version are compared with the hash table version that is based on the version from the paper. Even a manually optimized version performs worse than the hash table version.
The higher the function input parameter, the larger are the temporary tables of the CTEs. The larger a table is, the higher the benefit of using a hash table.
The presented method has enormous potential to improve query performance. Especially if the temporary tables are very large.

However, the presented method is not ideal. In order to ensure that the hash tables in the UDFs work without problems, the execution order must be enforced in some

places. This violates the declarative language paradigm of SQL. [14]
For this reason, this is only a proof of concept. A better implementation of the approach needs to be done. This work demonstrates that such an implementation is desirable.
The goal of this work is to show that the proposed method can bring a significant performance boost in the context of CTEs.

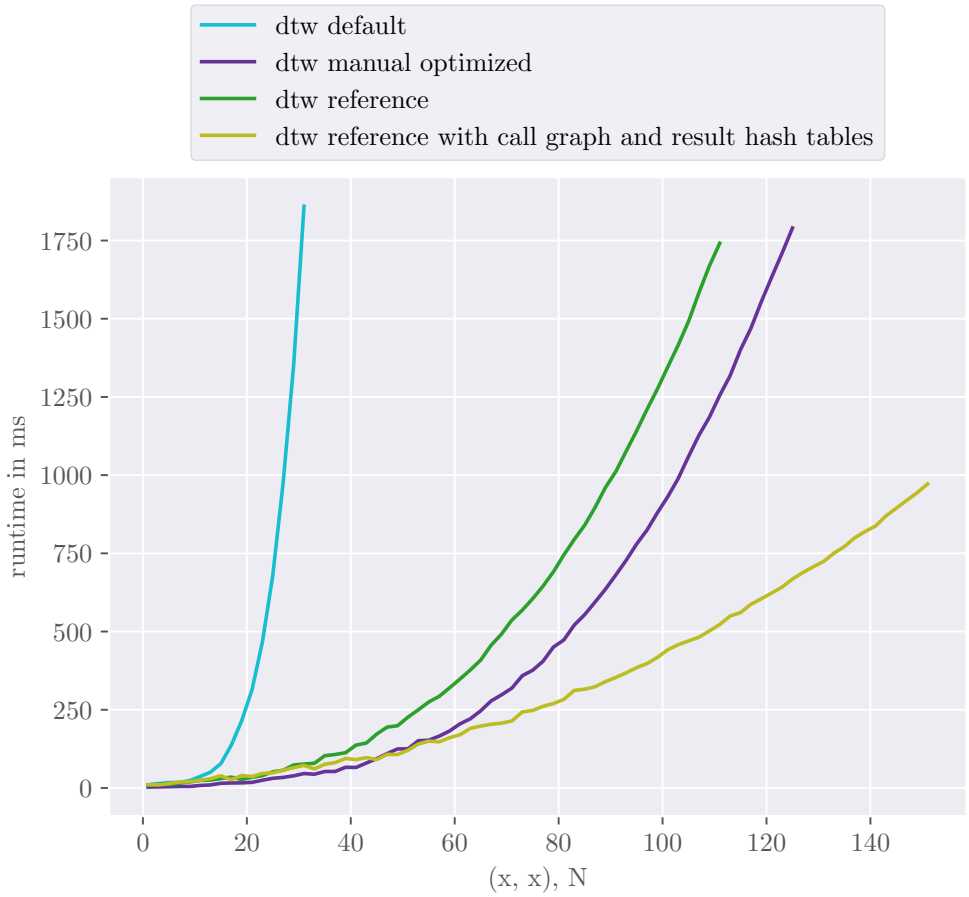


Figure 1.1: Runtime comparison of dynamic time warping (`dtw`) UDFs. The fsUDFs `dtw` default and `dtw` reference are based on the paper of Duta and Grust. The `dtw` reference version with hash tables is described in this thesis. A manual optimized version serves as comparison. The hash table version runs significantly faster for large inputs.

This thesis begins with the introduction in chapter 1. The related work section follows in chapter 2, where we take a closer look at the fsUDFs templates used in this thesis and a paper that uses hashes to improve database performance.

The extension that implements the hash tables is presented in Chapter 3. A run-

time comparison between sequential scan, index scan and hash table lookup is included.

In Chapter 4 we use dynamic time warping fsUDFs to describe two different approaches to use hash tables in a CTE to improve the performance.
These approaches get combined in subchapter 4.3, where both hash tables are used. The modifications by each hash table is independent from each other, which results in a full runtime benefit from both hash tables.
In subchapter 4.4 we describe a fsUDF version that uses a hash table for memoization. Since the hash tables are stored in working memory, the memoization is only available per session.

Chapter 5 applies the introduced hash tables onto other algorithm fsUDFs, that are generated with the same template. Depending on the table size, the hash tables have a varying performance boost. If the CTE that is boosted with a hash table is large, the hash tables performance of the fsUDF is increased. If the size is small, the hash tables have little or no positive impact on the performance.

In the last chapter 6 we conclude the thesis and give an outlook on future work.

2

Related work

In this thesis we use UDFs introduced in the paper *Functional-Style SQL UDFs With a Capital 'F'* of Duta and Grust [12]. The paper describes a method to convert simple recursive SQL UDFs into SQL recursive common table expressions with the goal of optimizing their runtime performance, while keeping them readable.
This is realized in two steps:

1. Constructing a call graph, that records the arguments of all recursive calls that the function would perform. The actual evaluations are done in the second step. The leaves of the graph represent non-recursive base cases of the function.

2. Traverse the call graph bottom up. All collected recursive calls are evaluated. The root on the top most level holds the result of the original function call.

The first step is implemented in the UDF in CTE `call_graph` and the second one in CTE `evaluation`. Several versions of the template are described in the paper using optimization techniques, such as reference counting and memoization.
With reference counting, rows of the CTE evaluation are dropped if they are no longer needed. This ensures that the work table size never exceeds a certain size and the runtime decreases.
The technique of memoization gives the possibility of storing results to a table and use them in subsequent function calls.
The UDFs in this thesis are based on a slightly updated version of the UDF templates introduced in the paper. Therefore runtime results of this thesis may differ from the results in the paper.

The method of applying hashes to improve database performance was used in the paper *Application of Hash to Data Base Machine and Its Architecture* of Kitsuregawa et al. [13].
They proposed a possible data base machine architecture called GRACE, which utilized

hash and sort to improve the performance of join and set operations. This was achieved by reducing the size of the data sets to be considered during join and set operations using hash operations, resulting in lower processing load.

3

Hash table extension

Before we can begin with implementing hash tables into PostgreSQL UDFs, we need to understand the hash table extension first. This chapter introduces the hash table extension with all used functions. In addition we compare the runtimes of a hash table lookup with the runtimes of sequential and index scans on temporary tables.

3.1 Implementation

We start with the different functions of the hash table extension:

- **Create hash table:**
  To be able to use a hash table, we need to create it first. This task is implemented with the function `prepareHT(tableId, numOfKeyColumns, colTypes)`. The first input parameter is the `tableId`, which is used to identify the hash table. Other functions can access a specific hash table by using its `tableId`.
  The next parameter defines how many columns the key consists of. The last input parameter defines the types and count of all columns.

- **Insert row:**
  The function `insertToHT(tableId, override, rowToInsert)` is used to insert rows into a hash table. The hash table `tableId` is used to refer a specific hash table and the number and type of columns must fit the referred hash table. The boolean override defines if a already existing entry can be overridden.

- **Get Row (returns a set of record):**
  The function `lookupHT(tableId, upsert, keyColumns)` gets a specific row of

the hash table with `tableId` and corresponding key columns of the row. The return type is a set of records. If the boolean upsert is true, a new entry is created if no existing entry matches.

- **Get Row (returns a record):**
  The function `lookupHTRecord(tableId, upsert, keyColumns)` is almost identical to the `lookupHT` function, but returns a record instead of a set of records.

The hash table extension uses the Postgres type `TupleHashTable` [8] to realize the hash tables. The function `prepareHT` uses the function `BuildTupleHashTableExt` [1] to create a `TupleHashTable`. The functions `insertToHT`, `lookupHT` and `lookupHTRecord` use the function `LookupTupleHashEntry` [5] to create or return tuples from the `TupleHashTable`.

In the following section we look into the runtime behaviour of the introduced hash table extension compared to table scans on temporary tables with and without index.

3.2 Runtime behaviour

In the following we compare the runtimes of three different methods to retrieve one specific row of a table, to compare the effect of the hash table extension to existing Postgres functionality. The methods are:

- sequential scan
- index scan
- hash table lookup

The experiment is performed with the following conditions: *(This and all following experiments where performed with PostgreSQL 13.4 running on a 64-bit Linux x84 host with 8 Intel Core^{TM} i7 CPUs clocked at 3.66 GHz and 64GB of RAM, 128MB of which were dedicated to the database buffer. Timings where averaged over 10 runs, with worst and best run times disregarded.)*
For the sequential and index scan a temporary table of size N is generated. The table consists of two columns `(id, value)` of type integer. Both columns contain rows with values from 1 to N. In case of the sequential scan, the table has an index on column `id`. An `ANALYZE` on the table ensures, that an index scan is used.
The hash tables consists of two columns of type integer, where the column `id` is the key. As in the previous tables, the table contains rows with values from 1 to N.

In case of the temporary table with and without index, following query is used to measure the runtime, see figure 3.1:

```
1  SELECT *
2  FROM table
3  WHERE table.id = N;
```

Figure 3.1: Query used to measure runtime of the sequential and index scan

In case of the hash table we use a hash table lookup, see figure 3.2:

```
1  SELECT *
2  FROM lookupHTRecord(tableId, false, N) AS table(id INT, value INT);
```

Figure 3.2: Query used to measure runtime of the hash table lookup

Temporary tables and hash tables are stored in the working memory. Therefore, the runtimes can be better compared with each other.

In figure 3.3 we can see that the runtime of the sequential scan increases with increasing table size. The reason is that all rows need to be considered in order to find all rows that fit the **WHERE** clause. The index scan and hash table lookup have a consistent low run time, because the searched row can be found faster due to the properties of the index and the hash table. The average runtime of a hash table lookup is $\Theta(1)$ [3] and PostgreSQL provides different index types to enhance query performance, such as B-trees that are suited for equality queries, which is the query type we use for the experiment [2] [4].

Inside UDFs we cannot make use of an index in a CTE scan. Hash tables provide the chance to simulate an index scan inside a UDF, as they are manual way to create a efficiently searchable structure that results in a similar performance to utilizing normal index scans. To enable the simulated index, all entries of the table need to be inserted in the hash table first.

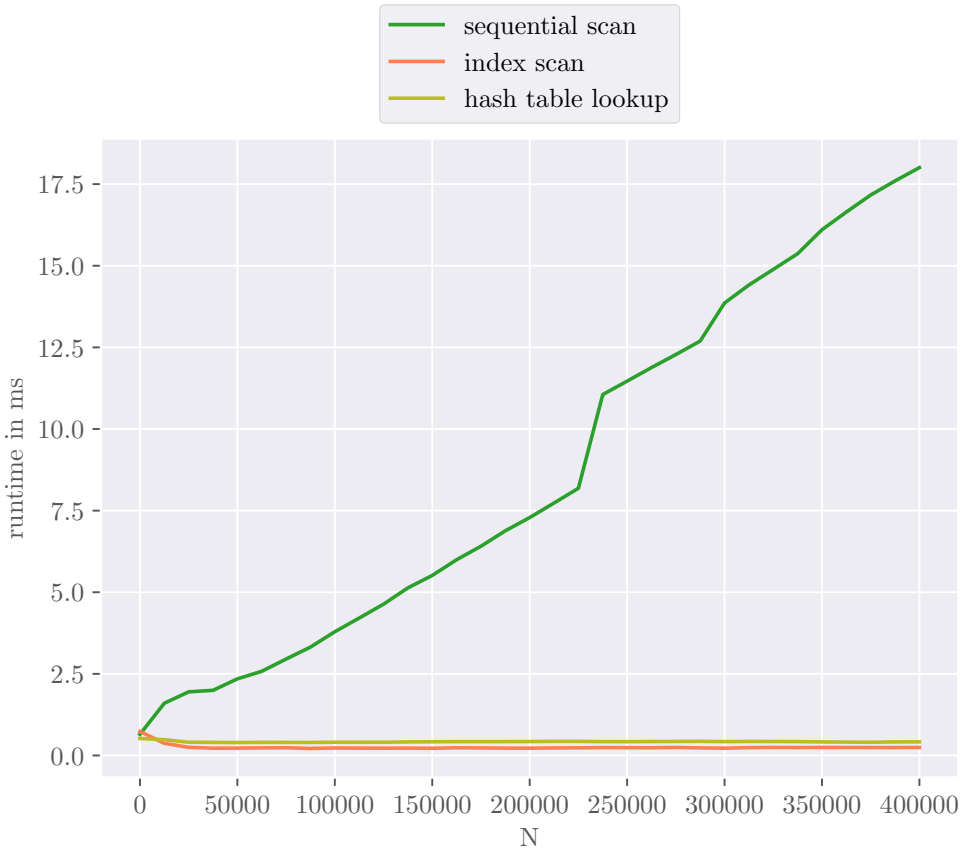


Figure 3.3: Runtime to retrieve one specific row in a temporary table or hash table size N

In the next chapter we use the hash table extension inside a fsUDF, which uses large tables, that only exist inside the UDF. Accessing specific entries in large tables is much faster with a hash table lookup instead of a sequential scan. Both hash table applications can be compared to a simulated index inside a UDF.
We will see that we have to break some laws of the declarative language in order to make the functions with hash tables work, because we need to enforce a certain execution order, so that the hash table already contains a specific row when it is queried.

4

Hash tables in dynamic time warping fsUDF

In this chapter we use the dynamic time warping fsUDFs, generated with the templates of Duta and Grust [12], to apply the idea of improving the runtime of CTEs with hash tables. There are three different templates we will consider: default, reference and memoization.

Dynamic time warping (`dtw`) is a three-fold recursive algorithm to find an optimal alignment between two given sequences [10]. It measures the similarity between two data sets X and Y by warping them along the time axis.

In chapter 4.1, the so called call graph hash table is described. This hash table simulates a CTE index on the `call_graph` CTE. After describing the implementation of the hash table in the default and reference `dtw` fsUDF, we consider the SQL plan changes and the resulting runtime results.

Another approach to use hash tables in the fsUDF is described in chapter 4.2. The result hash table stores intermediate results calculated in the `evaluation` CTE. In contrast to the call graph hash table, the result hash table is not static in size but grows over time.

In the following chapter 4.3, we use both hash table in the `dtw` reference fsUDF.
In the last chapter 4.4 we describe how the result hash table can be used to store intermediate results for subsequent executions.

4.1 Call graph hash table

In the first CTE `call_graph` a temporary table is constructed on which the CTE `evaluation` traverses. This table can reach a considerable size causing CTE scans to take a long time to return individual rows. As we have seen in chapter 3 a hash table lookup has a constant low runtime regardless of the table size. Therefore we will use the hash table extension to simulate a CTE index on the CTE `call_graph` to improve the runtime of the fsUDF.

We apply this method to two different template versions: default and reference. The reference version is based on the default version and additionally uses reference counting, which is a programming technique of counting the number of pointer references to each allocated object [7]. Improving the reference version is more relevant, because it performs much better than the default version. The insertion of the hash table is implemented based on the same concept in both versions.

First, we have to evaluate a suitable structure of the call graph hash table. For this, we analyze the structure of the call graph table with values of the call `dtw(2,2)`, see figure 4.1. [12]

in i | j	**site**	**fan out**	**out i | j**	**value**
(2, 2)	1	3	(1, 1)	□
(2, 2)	2	3	(1, 2)	□
(2, 2)	3	3	(2, 1)	□
(1, 1)	1	3	(0, 0)	□
(1, 1)	2	3	(0, 1)	□
(1, 1)	3	3	(1, 0)	□
(1, 2)	1	3	(0, 1)	□
(1, 2)	2	3	(0, 2)	□
(1, 2)	3	3	(1, 1)	□
(2, 1)	1	3	(1, 0)	□
(2, 1)	2	3	(1, 1)	□
(2, 1)	3	3	(2, 0)	□
(0, 0)	□	0	(0, 0)	0
(0, 1)	□	0	(0, 1)	∞
(1, 0)	□	0	(1, 0)	∞
(0, 2)	□	0	(0, 2)	∞
(2, 0)	□	0	(2, 0)	∞

Figure 4.1: Tabular call graph generated by the CTE `call_graph` for `dtw(2,2)`, where □ stands for a null value.

The choice of the key of the hash table is determined by how the hash table is accessed. In figure 4.2, you can see an original access to the call graph table. Here a join is performed on the columns `out_i`, `out_j`. From this we conclude that these two columns must be the keys of the hash table.

```
1  SELECT  ...
2  FROM    call_graph AS g, e
3  WHERE   (g.out_i, g.out_j) = (e.in_i, e.in_j)
```

Figure 4.2: Snippet from the `dtw` reference fsUDF: Reading of entries from the CTE `call_graph`. We conclude, that the columns `out_i`, `out_j` are suitable keys for the call graph hash table.

Looking at the values of the table in figure 4.1, we see that the key of the hash table is not unique. The uniqueness of the key is a mandatory requirement for a hash table. To restore the uniqueness, we combine all entries with the same key in an array.
Which means the return value of an access to the hash table is an array. With unnest, this array is converted into a table, in which we can get the queried value.
In order for the `site`, `in_i` and `in_j`, `fan_out` and `val` columns to fit into one array entry, the composite type **`CALLGRAPHVAL`** is created.

The final structure is therefore a hash table based on the two keys `out_i`, `out_j` pointing to an array of type **`CALLGRAPHVAL`**.

As the next step we implement the call graph hash table in the `dtw` default and reference fsUDFs.
In figure 4.3 the structure of `dtw` default and reference is shown. A complete version of all `dtw` functions and their versions with hash tables can be found in the appendix, see section 6.
First, the hash table is created, then filled, and finally used in the `evaluation` CTE to traverse over the call graph.

```
1  CREATE FUNCTION f(args)
2  RETURNS DOUBLE PRECISION
3  AS $$
4  SELECT prepareHT(CG_HT_ID, 2,     NULL::INT,
5                                    NULL::INT,
6                                    NULL::CALLGRAPHVAL[]);
7  WITH RECURSIVE
8      call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
9      (...),
10     fill_hashtable AS
11     (...),
12     base_cases(in_i, in_j, val, ...) AS
13     (...),
14     evaluation(in_i, in_j, val, ...) AS
15     (...)
16 SELECT  e.val
17 FROM    evaluation AS e
18 WHERE   (e.in_i, e.in_j) = ((dtw.args).i, (dtw.args).j);
19 $$ LANGUAGE SQL STABLE
20                 STRICT;
```

Figure 4.3: Simplified function body of `dtw` default and reference with call graph hash table, where red is used to highlight the changes compared to the version without hash tables.

At the beginning of the UDF the `prepareHT` function is used to create the hash table. Two integer values are keys and point to a **`CALLGRAPHVAL`** array. `CG_HT_ID` is the `tableId` of the call graph hash table.
The creation of the call graph is the first CTE of the UDF and remains unchanged. The CTE `fill_hashtable` is a new component and serves the purpose of populating the hash table. To do this, all values of the call graph table are read and entered into the hash table.
To ensure that that `fill_hashtable` has been fully executed, and therefore all values inserted, before it is accessed for the first time, a **`SELECT COUNT`**`(*)` is performed on the `fill_hashtable` CTE as a subquery in the `base_cases` CTE, see figure 4.4. Otherwise, a searched entry might not be found, because PostgreSQL does not enforce that all rows, or at least the searched row, are entered in the hash table before reading rows. Thus the function will not calculate the correct result reliably or will not terminate. An execution order must be guaranteed to ensure that the function runs correctly. This means that the declarative language paradigm of SQL is violated, which defines the logic of a computation without describing its control flow [14]. The necessity for this execution order trick is the biggest drawback of the current hash table implementation in the context of CTEs.

```
1  base_cases(in_i, in_j, val) AS (
2      SELECT  g.in_i, g.in_j, g.val
3      FROM    call_graph AS g,
4              (SELECT COUNT(*) FROM fill_hashtable) AS _
5      WHERE   g.fanout = 0
6  ),
```

Figure 4.4: CTE `base_cases` of `dtw` default with call graph hash table, where red is used to highlight the changes.

The routine `base_cases` of the default and reference version are different in the number of references of the call graph table.
In the default version, the call graph is referenced once and only entries where `fanout=0` is true are searched. The selected key of the hash table is not used here to access entries in the call graph table. For that reason, the table is not replaced with the hash table in this case.

```
1  base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS (
2      SELECT  g.in_i, g.in_j, g.val,
3              g_ref.in_i, g_ref.in_j, g_ref.site, g_ref.fanout
4      FROM    call_graph AS g,
5              (SELECT COUNT(*) FROM fill_hashtable) AS _,
6              lookupHTRecord(CG_HT_ID, false, g.in_i, g.in_j)
7                  AS ht(out_i INT, out_j INT, cgval CALLGRAPHVAL[]),
8              unnest(ht.cgval) AS g_ref
9      WHERE   g.fanout = 0
10       AND   (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
11 ),
```

Figure 4.5: CTE `base_cases` of `dtw` reference with call graph hash table, where red is used to highlight the changes.

In the case of `dtw` reference `base_cases` (see figure 4.5), the call graph table is referenced twice to perform a self join. One reference to the table can be replaced with a hash table call. This eliminates the **`WHERE`** clause with the self join. By doing this we can replace a CTE scan with a hash table lookup, which aims to improve the performance.

In the `evaluation` CTE, the call graph table is traversed. In both versions only calls like in figure 4.6 occur in the `evaluation` CTE.

```
1  SELECT r.in_i, r.in_j, r.val, g.in_i, g.in_j, g.site, g.fanout
2  FROM returns AS r,
3       call_graph AS g
4  WHERE (r.in_i, r.in_j) = (g.out_i, g.out_j)
```

Figure 4.6: Snippet of `dtw` reference: Access to entries of call graph table.

With the same method as in `base_cases` of `dtw` reference, these calls can be replaced with a hash table call and a subsequent unnest, see figure 4.7.

```
1  SELECT  r.in_i, r.in_j, r.val, g.in_i, g.in_j, g.site, g.fanout
2  FROM    returns AS r,
3          lookupHTRecord(CG_HT_ID, false, r.in_i, r.in_j)
4             AS ht(out_i INT, out_j INT, cgval CALLGRAPHVAL[]),
5          unnest(ht.cgval) AS g
```

Figure 4.7: Snippet of `dtw` reference with call graph hash table: Access to entries of call graph hash table

By using the call graph hash table to access the content of the `call_graph` CTE we implement a simulated CTE index. This is implemented by inserting the rows of the `call_graph` CTE into a hash table and replace all suitable occurrences in the following CTEs. The drawback is the usage of the trick to enforce the execution order.

In the next section we will look at the plan changes for both `dtw` versions produced by the insertion of the call graph hash table.

4.1.1 SQL plan changes

In this chapter we examine the impact on the SQL plan of replacing calls on the `call_graph` CTE with lookups on the call graph hash table. Only differences between the plans will be discussed. The `dtw` versions default and reference are treated separately from each other.

Call graph hash table in the `dtw` default template

The SQL plan of the `dtw` default version is extended by the CTE `fill_hashtable`, see figure 4.8. It performs a CTE scan on the call graph table and aggregates it. As already mentioned, the CTE is used to insert rows from the call graph table into the hash table.

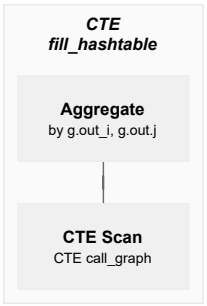


Figure 4.8: The CTE `fill_hashtable` forms a new block in the SQL plan.

The CTE `base_cases` is embedded in the `evaluation` CTE in both versions. The original `base_cases` consists of a single CTE scan of the call graph table. In the hash table version it contains an additional CTE scan of the CTE `fill_hashtable`, see figure 4.9.

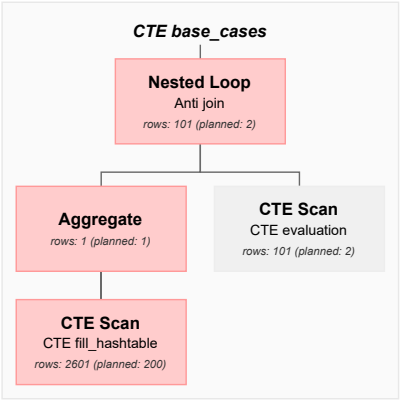


Figure 4.9: The SQL plan of CTE `base_cases` in `dtw` default of the hash table version. Red marks the changes compared to the original SQL plan.

The next change in the SQL plan is in the `evaluation` CTE, in which the CTE `returns` is included.
The CTE `returns` differs in several points, see figure 4.10. In the version without hash table the CTE is embedded in the CTE `evaluation`. In the hash table version, this CTE forms its own block.
The CTE `returns` contains a subquery scan at the top level, whose right child contains no changes. The left child contains several changes.
The Nested Loop is replaced by a Hash Join. In the original version both join partners have low estimated row counts. The Hash Join in the hash table version is chosen, because a higher row count is estimated from the left join partner.
For a similar reason a Hash Join is replaced by a Nested Loop, because the estimated row count decreased.
An additional Nested Loop is added, because of the replacement of the CTE Scan of the CTE `evaluation` with a Function Scan, which is the hash table lookup. This additional Nested Loop is necessary to apply the Function Scan with unnest.

The SQL plan changes only take place, where the hash table was built in. The CTE `returns` now forms a new block and Hash Joins have been replaced with Nested Loops and vice versa. Otherwise, no structural changes can be observed. Therefore, all the runtime changes can be attributed to the hash table and not to a plain plan improvement.

In the next section, we discuss the SQL plan changes between the `dtw` reference versions with and without the call graph hash table.

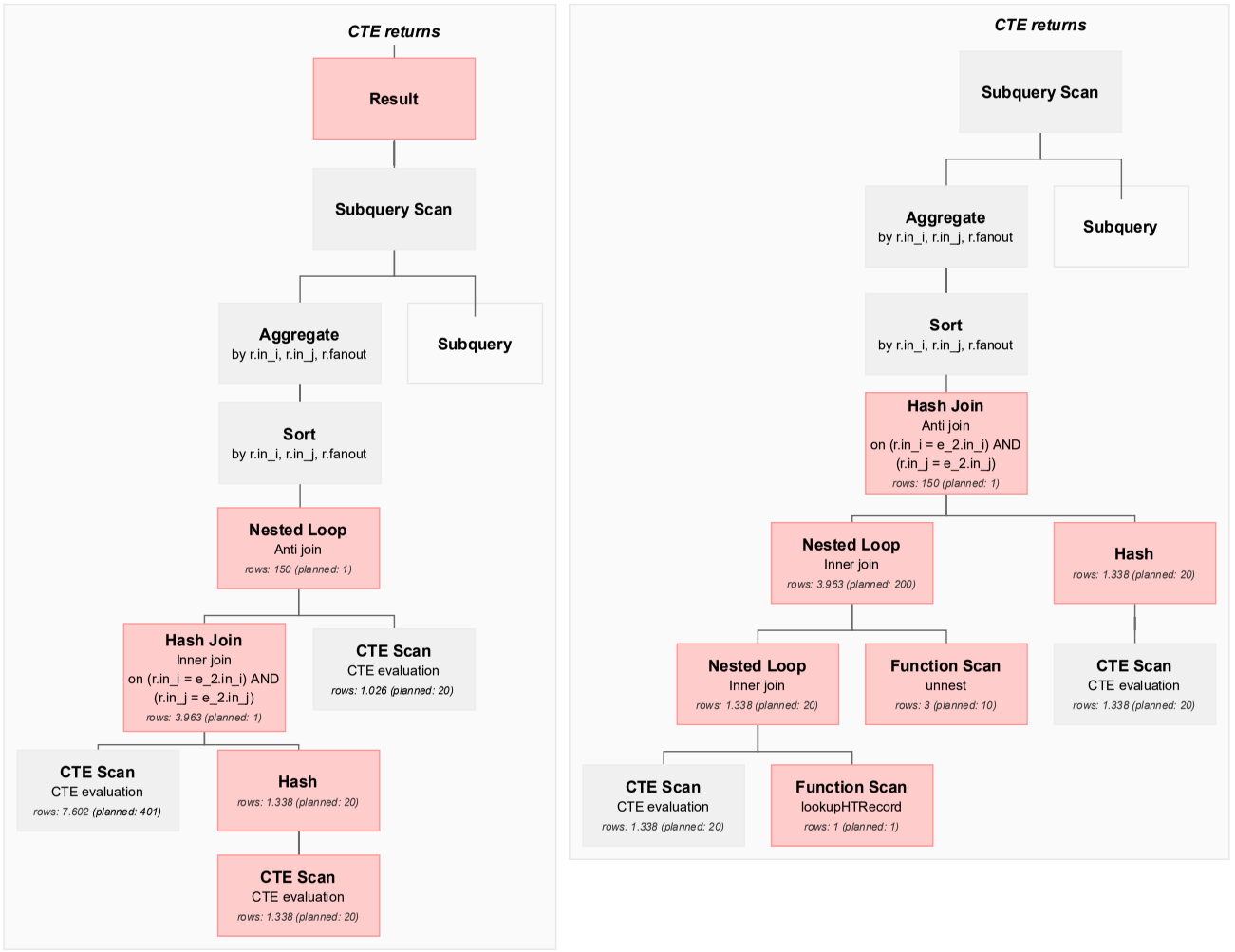


Figure 4.10: The SQL plan of CTE returns of `dtw` reference: On the left side the version without call graph hash table, on the right side the version with call graph hash table. The differences are highlighted in red

Call graph hash table in the `dtw` reference template

The SQL plan changes of the `dtw` reference version with and without call graph hash table differ in the CTEs `fill_hashtable`, `base_cases` and `evaluation`.

As in the default version, the CTE `fill_hashtable` is added as a new block.

The CTE `base_cases` is in the hash table version not embedded in the `evaluation` CTE. The plan of the CTE `base_cases` got more complex, see figure 4.11 and 4.12. Due to the addition of a CTE Scan of the `fill_hashtable`, a new Nested Loop is added. The exchange of one CTE scan of the `call_graph` with the hash table lookup, added two Function Scans and one Nested Loop. No Hash Joins are used in the new plan, because only low row counts are estimated.

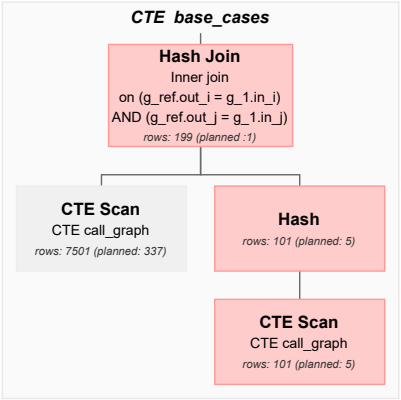


Figure 4.11: SQL plan of CTE `base_cases` in `dtw` reference without call graph hash table.

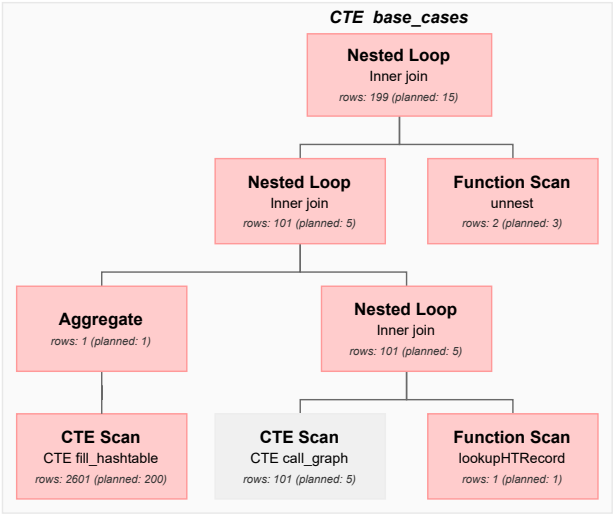


Figure 4.12: SQL plan of CTE `base_cases` in `dtw` reference with call graph hash table.

In figures 4.13 and 4.14, the plans of the CTE `evaluation` before and after adding the call graph hash table are shown. As previously mentioned, the CTE `base_cases` is no longer embedded in the hash table version.
The original version uses a Hash Join, because a high number of rows are estimated. In contrast, only Nested Loops are used in the hash table version, because low row counts are planned. The CTE Scan of CTE `call_graph` is replaced by two Function Scans and one additional Nested Loop, which perform the hash table access.
Another change is the swap of the join partners. Originally the CTE Scan on the CTE `returns` is the right join partner and in the hash table version it is the left one, the planned number of rows is no longer lower than the row number of the join partner. Therefore it can be the left join partner.

Like the plan changes in `dtw` default, the plan changes in `dtw` reference can be attributed to the implementation of the call graph hash table. No unexpected plan changes have occurred that can additionally affect the runtime.

In the next section, the runtime results of the `dtw` default and reference versions with and without call graph hash table are analyzed.

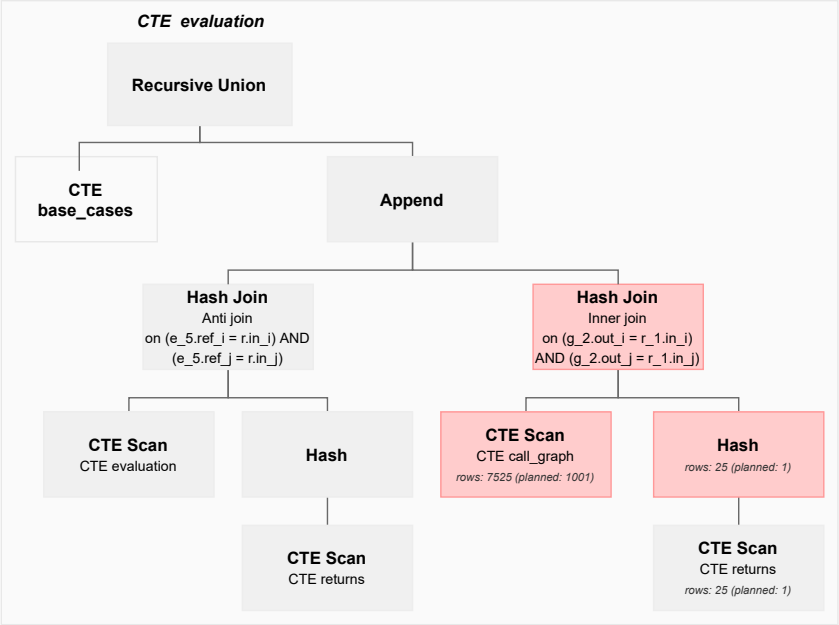


Figure 4.13: SQL plan of CTE `evaluation` in `dtw` reference without call graph hash table, where red is used to highlight the changes.

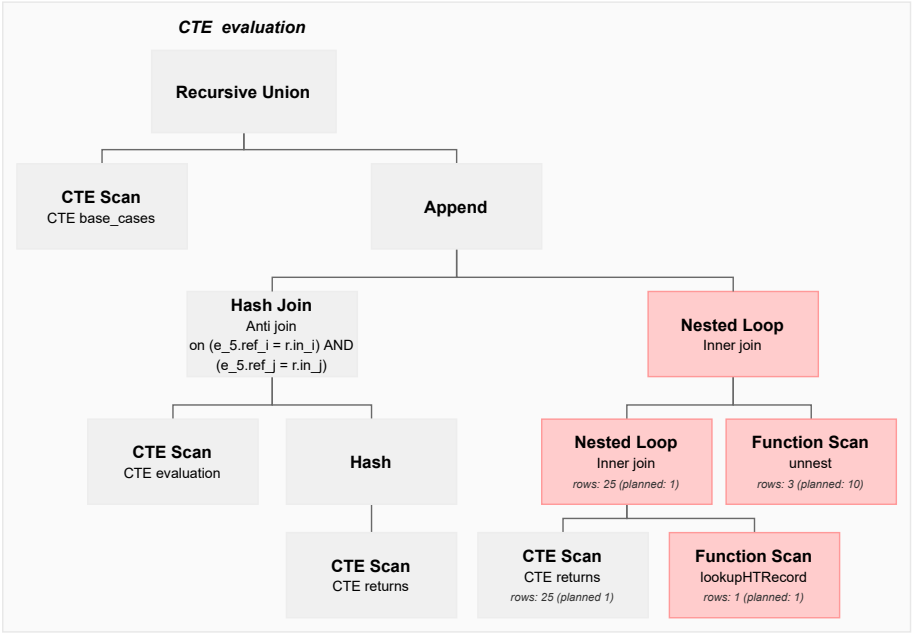


Figure 4.14: SQL plan of CTE `evaluation` in `dtw` reference with call graph hash table, where red is used to highlight the changes.

4.1.2 Runtime results

In this section we look at runtime results of the `dtw` default and reference versions with and without call graph hash tables. First we describe how the measurements were created and continue with the actual results.

Experiment setup

The `dtw` function runs on the input tables X and Y. The tables are filled with N random entries. The input parameters of the function are `dtw(i, i)` where i is chosen to be set to N, which means the complete tables X and Y of size N are considered.
In the appendix, the complete setup of the function input tables is shown.
To determine a runtime for a `dtw` call with fixed input parameters, ten runtimes are measured. Then the minimum and the maximum are removed and the average is calculated.

Figure 4.15 shows that different runtimes are measurable for different sizes of the input tables X and Y. Depending on the input table sizes, PostgreSQL uses different SQL plans. Thus, instead of an index scan, a sequential scan is performed on the input tables, resulting in a longer runtime.
For small N, sequential scans are used to access the tables X and Y. For high N values all sequential scans are replaced by index only scans. The value range in between, where N is about 190-410, sequential scans and index only scans are part of the plan.
To enforce a uniform plan, that uses index only scans, we set the parameter `random_page_cost` from 4 to 0, which defines the planner's estimate of the cost of a non-sequentially-fetched disk page. [6] So the system prefers an index only scan already at smaller input table sizes. We use this setup for all measurements of `dtw` functions.

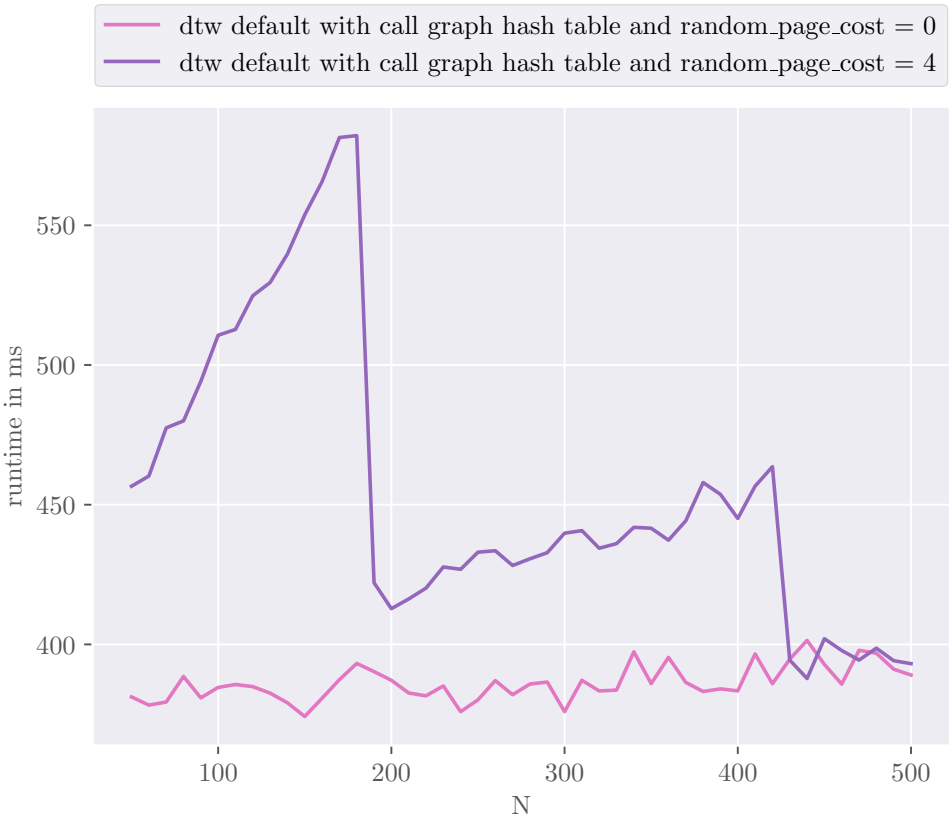


Figure 4.15: Runtime comparison of `dtw` default version with call graph hash table with different `random_page_cost` set. The function parameter are always `dtw(50,50)` and the table size N increases. The three distinct sections of the purple curve are from left to right: using mostly sequential scans, using a mix of index and sequential scans, and finally using only index scans which leads to the best performance.

Runtimes

With the described experiment setup we get following runtime results, which are shown in figure 4.16. The versions that use the call graph hash table are faster than the initial versions.

The `dtw` default version benefits significantly from the use of the hash table. The slope of the running time curve has decreased significantly, getting closer to the running time curve of `dtw` reference.
Also the generally faster reference version benefits from the hash table greatly, but to a lesser extent than the `dtw` default version. The runtime reduction increases with increasing parameter size. For `dtw(120, 120)`, the runtime of the hash table version reduces to 63% of the original runtime.

In chapter 5, we will observe that the degree of runtime improvement depends on the number of rows in the CTE. The larger the table, the greater the advantage of the hash table, since the runtime of CTE scans is higher on large tables.

Based on the runtime results in case of `dtw`, it is clear that the simulated `call_graph` CTE index has achieved the desired increase in performance. The runtimes improved visibly in both cases.

In the next chapter 4.2, we focus on the second hash table, which offers another possibility for runtime optimization.

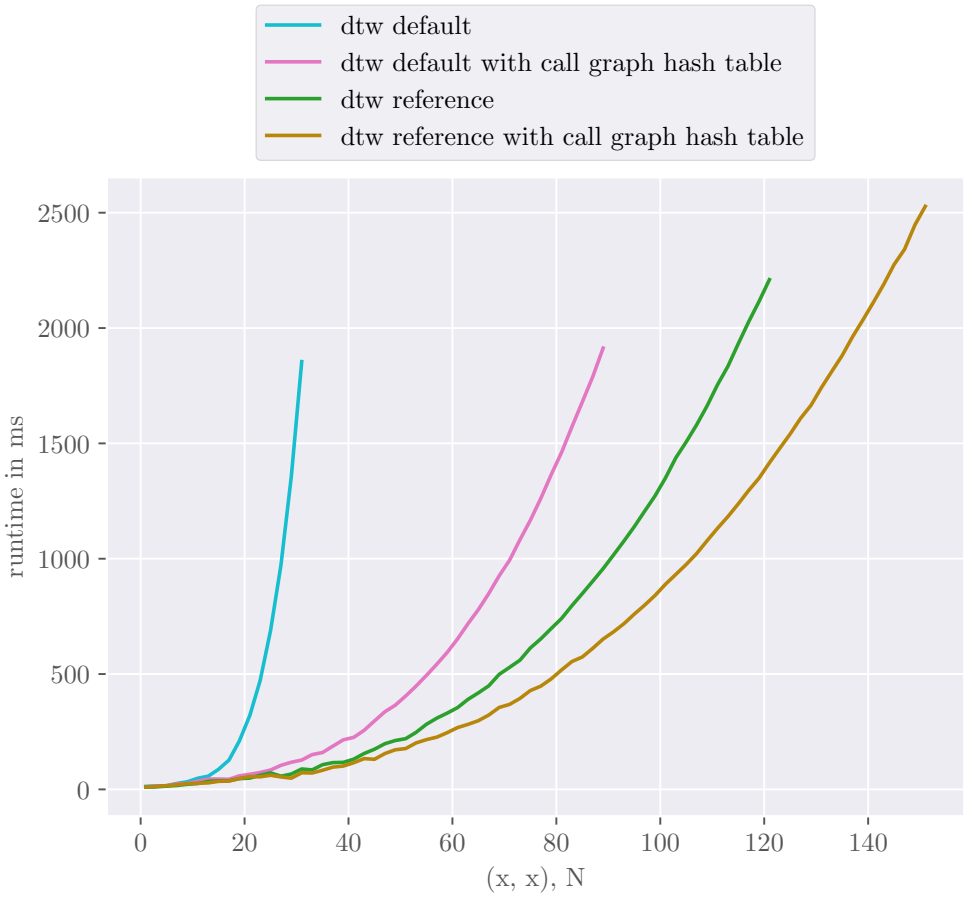


Figure 4.16: Runtime comparison of `dtw` default and reference versions with and without call graph hash table. The function is called with `dtw(x,x)` and N as table size of input tables X and Y.

4.2 Result hash table

In the `dtw` reference version a further place provides the opportunity to use a hash table. In the `evaluation` CTE, intermediate results of the calculation are stored and at the next recursive call specific values are extracted. It is a constantly changing intermediate result table, which is used at the end of the function to call the output result.
We compared the call graph hash table of the previous chapter with a static CTE index. This is not the case for the result hash table, because it is not filled with values once and then only used for reading, instead in the process of the `evaluation` CTE it is constantly updated by writing new values into it and at the same time it is used for reading.
The result hash table can be compared with a dynamic CTE index, that is constantly updated in each recursive step.
Accesses on the hash table are cheaper than CTE scans, which is why using the hash table for storing intermediate results improves the runtime. Like in the call graph hash table once again we have to enforce the execution order at some points to ensure the correct execution of the UDF.

In the next section we show which modifications to the `dtw` reference version are introduced to store the intermediate results in the result hash table. This version of the function does not include the changes made by implementing the call graph hashtable in the previous chapter. In chapter 4.3 we will use both hash tables in one function. The complete function is shown in the appendix, see section 6. First of all, we consider the structure of the hash table.
In table 4.17 a part of the content of the result table after the complete execution of the `evaluation` CTE can be seen. Only the columns of the table are shown, which are necessary for the calculation and storage of the intermediate results. We will create the hash table only for these three columns. The other columns of the CTE are still necessary for a correct function execution. The hash table serves as a dynamic index on the selected columns, which improves the performance.

i	j	value
0	0	0
0	1	∞
1	0	∞
1	1	0
1	2	1
2	1	0
2	2	1

Figure 4.17: Selected columns of the tabular result table generated by the CTE `evaluation` for `dtw(2,2)`

The input parameters, as they are used when calling the function, are mapped to a value, which may also be null value. It follows two integer values point to a double precision value. The keys of the hash table are unique, therefore no array is necessary for the result hash table.

Examining the modified structure of the `dtw` reference version in figure 4.18, at the beginning of the function there is the creation of the hash table with the stated structure.

```
1  CREATE FUNCTION f(args)
2  RETURNS DOUBLE PRECISION
3  AS $$
4  SELECT prepareHT(R_HT_ID,  2,    NULL::INT,
5                                   NULL::INT,
6                                   NULL::DOUBLE PRECISION);
7  WITH RECURSIVE
8      call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
9      (...),
10     base_cases(in_i, in_j, ref, ref_site, ref_fanout) AS
11     (...),
12     evaluation(in_i, in_j, ref, ref_site, ref_fanout) AS
13     (...)
14 SELECT  result_ht.val
15 FROM    (SELECT COUNT(*) FROM evaluation) AS _,
16         lookupHTRecord(R_HT_ID, false, (dtw.args).i, (dtw.args).j)
17            AS result_ht(i INT, j INT, val DOUBLE PRECISION);
18 $$ LANGUAGE SQL STABLE
19                 STRICT;
```

Figure 4.18: Function body of the `dtw` reference version with result hash table, red marks the changed parts compared to the version without the result hash table.

In the `call_graph` CTE the base cases for the evaluation are calculated, which are normally inserted via the `base_cases` CTE in the intermediate results table. We use the function `insertToHT` to insert these base cases directly inside the the `call_graph` CTE into the hash table.
The `base_cases` CTE remains unaffected. Although we insert new values into the hash table in the `call_graph` CTE, there is no need to use the trick to ensure that all values are inserted before the hash table is used in the `evaluation` CTE. It can be safely assumed that the `call_graph` CTE has been fully executed, since the `base_cases` CTE contains a self join onto the `call_graph` CTE and thus the complete table is traversed to return a row.
In the `evaluation` CTE we exchange the accesses on the evaluation table with hash table lookups, see figure 4.19. It follows, that the intermediate results are read from the result hash table and the result of the calculation is written directly back into the hash table.

```
1  insertToHT(R_HT_ID, true, i, j,
2    (SELECT CASE
3            ...
4            ELSE (SELECT abs(Z.x - Z.y)
5                         + LEAST((SELECT result_ht.val
6                                  FROM   lookUpHTRecord(
7                                                R_HT_ID, false,
8                                                go.in_i - 1,  go.in_j - 1)
9                                                AS result_ht(i, j, val)),
10                                 (SELECT result_ht.val
11                                  FROM   lookUpHTRecord(
12                                               R_HT_ID, false,
13                                               go.in_i - 1, go.in_j)
14                                               AS result_ht(i, j, val)),
15                                 (SELECT result_ht.val
16                                  FROM   lookUpHTRecord(
17                                               R_HT_ID, false,
18                                               go.in_i, go.in_j - 1)
19                                               AS result_ht(i, j, val))), ...
```

Figure 4.19: Simplified version of the usage of the result hash table in the `evaluation` CTE. Red marks the modified parts compared to the version without the result hash table. A new entry of the result table is computed by looking up multiple values in the result hash table. The result of the computation is directly written into the result hash table.

At the end of the `dtw` reference function the final result of the function is called directly from the hash table. At this point we must again enforce that the `evaluation` CTE was executed completely, which is the same workaround used in chapter 4.1, see figure 4.18.

As the result of the SQL code modifications, we get a `dtw` reference version that uses a hash table to store and retrieve the intermediate results, as well as the final result, in it.

In the next section we look at what impact the SQL code changes have on the SQL plan and in the subsequent section we compare the runtimes of the `dtw` reference version with and without the result hash table.

4.2.1 SQL plan changes

The SQL plan has changed in the `evaluation` CTE and in the main query part. The plan of the `call_graph` routine remains unchanged, although the CTE is modified. The insertion of values into the hash table has not caused any visible plan changes at this point.
In the final part of the `dtw` reference function the result from the `evaluation` CTE is called. Originally the plan consisted only of a CTE scan on the `evaluation` routine, see figure 4.20.
In the hash table version, a Function Scan is used to return the result of the function, see figure 4.21. A CTE Scan on CTE `evaluation` is still part of the plan, because a **`SELECT COUNT`**`(*)` **`FROM`** `evaluation` must performed to ensure the hash table contains all entries.

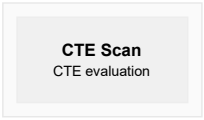


Figure 4.20: Plan of final block in `dtw` reference without result hash table

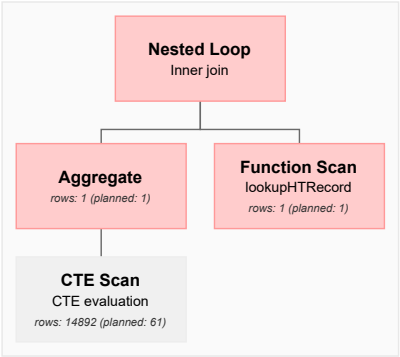


Figure 4.21: Plan of final block in `dtw` reference with result hash table

More changes are in the CTE `evaluation`, see figures 4.22 and 4.23.
In the original plan, the red marked Hash join performs the join between `evaluation` and `returns` CTE. In the new plan, instead a CTE Scan and a Function Scan, that is a lookup on the hash table, is performed to do this job. The Hash join is no longer necessary. In addition a Nested Loop is needed. The left join partner is a CTE Scan on CTE `returns` with an Aggregate. This is again the trick we need to enforce the execution order, which uses a **`SELECT COUNT`**`(*)` on `returns`.

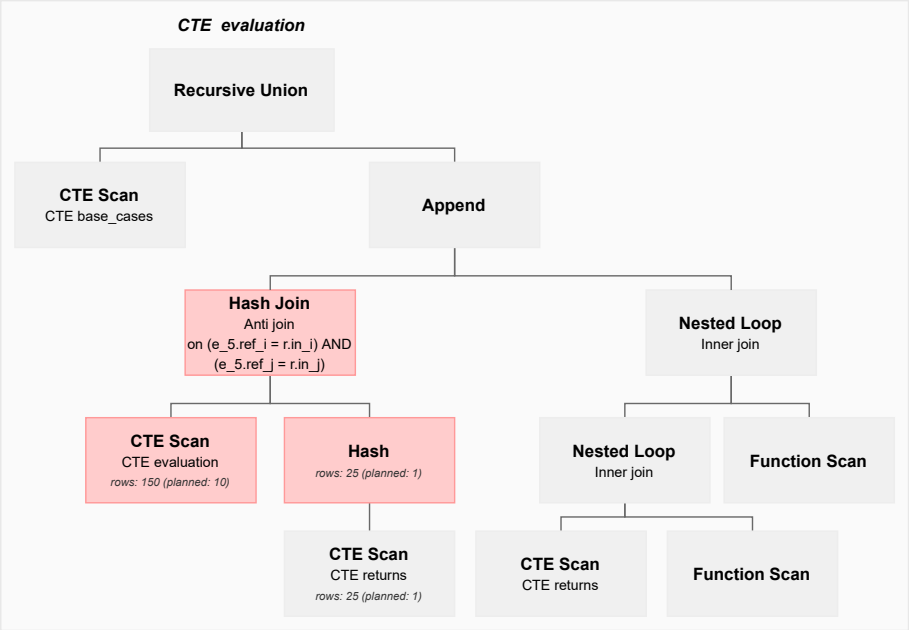


Figure 4.22: SQL plan of CTE `evaluation` of `dtw` reference without result hash table.

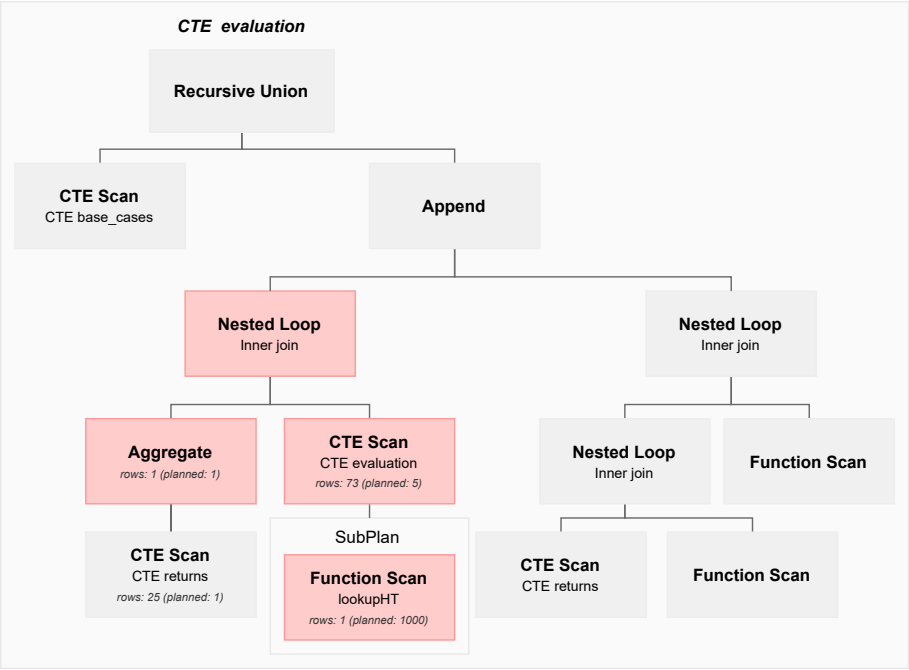


Figure 4.23: SQL plan of CTE `evaluation` of `dtw` reference with result hash table.

The plan of CTE `returns` differs in several places. All CTE scans on table `evaluation` are replaced with a Function Scan with function `lookupHTRecord`. This is where the result hash table is used to call previously calculated results. The storage of the calculated results in the result hash table is not visible in the plan.

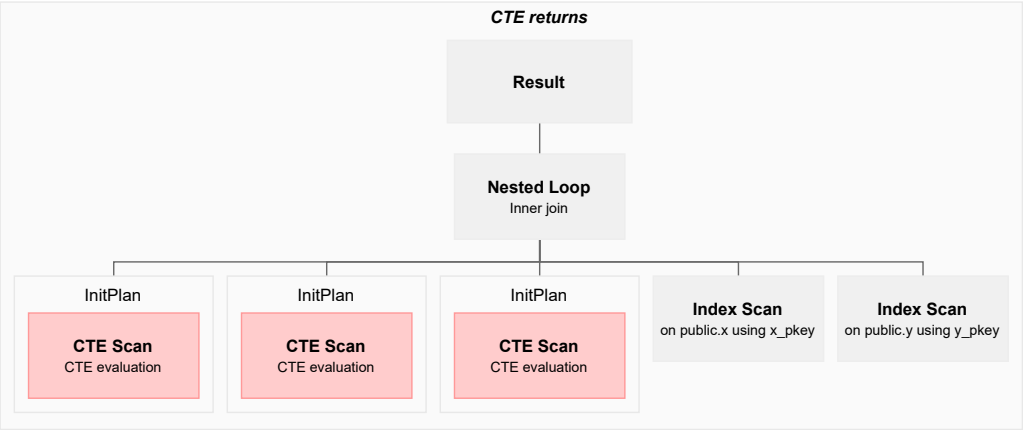


Figure 4.24: SQL plan of CTE returns of `dtw` reference without result hash table.

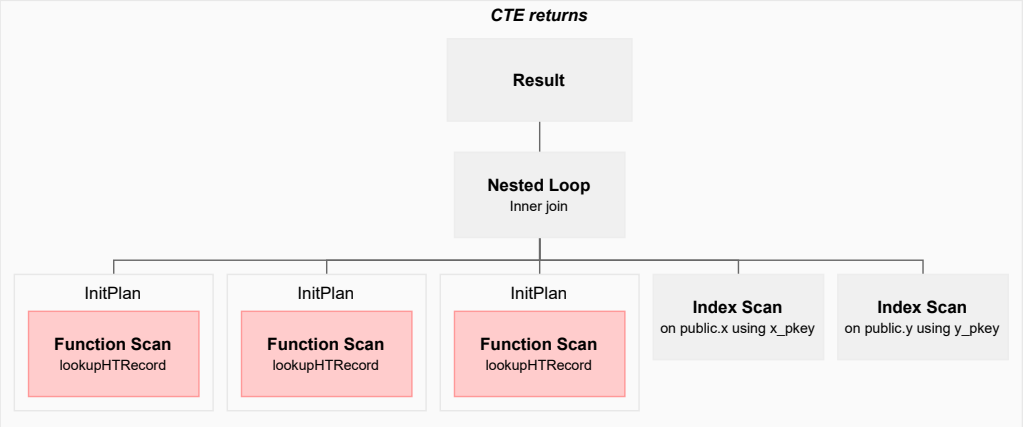


Figure 4.25: SQL plan of CTE returns of `dtw` reference with result hash table.

The SQL plan changes of the `dtw` reference version that uses the result hash table can be directly linked to the insertion of the hash table. There are no unexpected plan changes, that cannot be explained by the usage of the hash table. In addition, we observed that inserting values into the hash table, does not visibly change the plan.

How the `dtw` reference version with result hash table performs is discussed in the following section.

4.2.2 Runtime results

The runtime measurements were made under the same settings as described in chapter 4.1.2.
Figure 4.26 shows the runtime results of the basic `dtw` reference version, the `dtw` reference version with result hash table and for comparison the `dtw` reference version with call graph hash table.
The runtime of the version with result hash table has improved significantly compared to the original version. Replacing CTE scans with hash table look ups shows its effect. There is a runtime advantage of the simulated dynamic CTE index over using no index. This version has almost the same runtime reduction as the version with the call graph hash table. Generally the versions with hash tables have a lower execution time than the version without hash table. This runtime advantage is especially visible for large input tables X and Y, meaning high N values. This allows the calculation of larger input values, which previously had a runtime too high to be feasible.

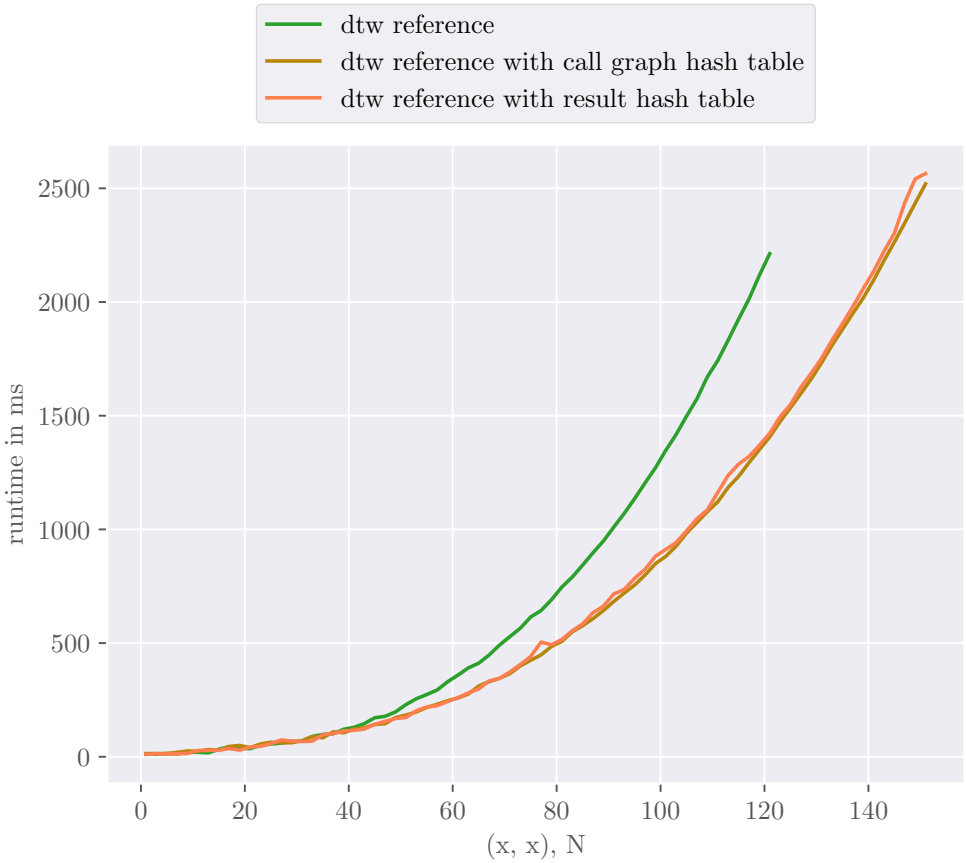


Figure 4.26: Runtime of `dtw` reference with no hash table, call graph hash table and result hash table. The versions with hash table behave similarly.

In the following chapter 4.3 we describe how both hash tables behave in a combined version.

4.3 Call graph and result hash table

In the previous chapters it was shown how hash tables can be used to improve the query performance by implementing a simulated index on the call graph and on the result table in the `dtw` reference version. Both modifications improve the runtime compared to the version without hash tables.
In this chapter use both hash tables in the `dtw` reference function. The method how the two hash tables are inserted into the function is the same as the method described in the corresponding chapters. There are no parts that are changed in both chapters. Therefore, the adjustments can be adopted exactly.

In the following, we examine the impact on the SQL plan and the runtime of the function when the call graph and result hash tables are used simultaneously in the `dtw` reference function.

4.3.1 SQL plan changes

Examining the SQL plan changes of the version with both hash tables, we find no surprises. By combining the plan changes of `dtw` reference caused by the insertion of the call graph and result hash tables (see chapter 4.1.1 and chapter 4.2.1), the SQL plan of the version with both hash tables is obtained. There are no unexpected side effects and no mutual interferences.

4.3.2 Runtime results

The runtime results of the different `dtw` reference versions is shown in figure 4.27. The same evaluation conditions as in chapter 4.1.2 were used to create the measurements.

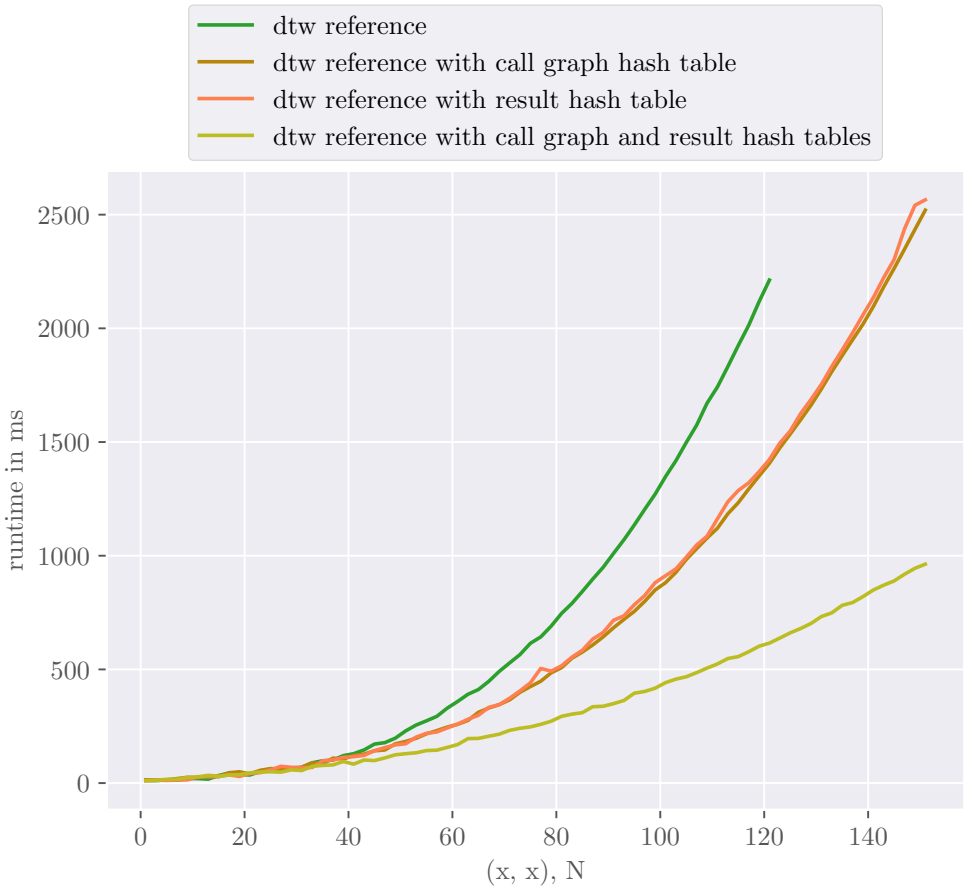


Figure 4.27: Runtime of `dtw` reference with no hash table, call graph hash table, result hash table and both hash tables.

The `dtw` reference version that uses both hash tables has the lowest runtime of all versions considered.
If we calculate the runtime reductions of the versions with only one hash table and subtract them from the base version, we get the runtime of the combined version. This means that the runtime improvements of the individual versions are independent of each other. This also matches the observations from the SQL plans, since the individual changes do not have any overlap or interference.
For example `dtw(120, 120)` has a runtime of 2156 ms. The runtime reduction of the single hash table versions are 784 ms and 743 ms. When we subtract these values from the runtime of the base version, we get 629 ms. Which is very close to the actual runtime

result of 608 ms.
The regarded exponential runtime of the combined version has a lower slope than the other curves. This means that the performance improvement itself increases exponentially. This means we obtain an immense improvement in performance by using both hash tables.
Figure 4.28 shows the table sizes of the call graph table and the evaluation table. These tables are enhanced by the usage of the hash tables call graph and result.
The result hash table represents only a subset of the evaluation table. It has fewer columns, but the number of entries is the same. The evaluation table is used for the calculation before it reaches the size shown in the figure.
We can deduce from the table sizes how big the advantage of the hash tables will be. In the case of `dtw`, the evaluation table is larger than the call graph table. Nevertheless, the two hash tables produce about the same runtime reduction.
One factor is the use of the evaluation table before it reaches full size. The larger the table, the greater the advantage of the hash table, because CTE scans have higher runtimes at large table sizes. Another factor could be the number of calls on the tables. The more frequently the table is used, the greater the advantage of the hash table.

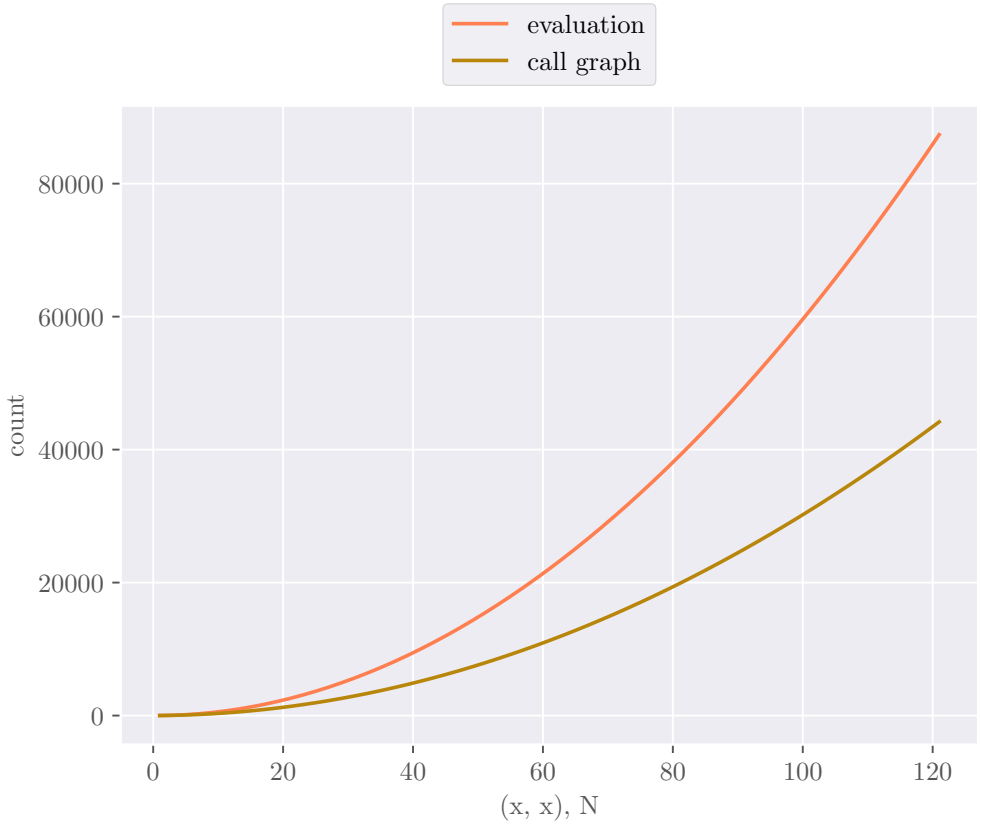


Figure 4.28: Table sizes of the call graph and evaluation CTE.

In chapter 5 we will see how different table sizes affect the runtime reduction due to the hash tables.
In the next chapter 4.4, we modify the memoization template version, so that it uses the result hash table for memoization.

4.4 Memoization inclusive

In this chapter we examine another version of the fsUDF functions: `dtw` with memoization. This version is based on `dtw` reference and additionally uses the possibility to store and reuse results from previous function calls [12].
This concept is implemented by:

- after the execution of the function, the results of the evaluation are saved in a previously created memoization table
- and in further executions the memoization table will be treated like additional base cases and can be used instead of recalculating the results

Performing multiple function calls based on the same input data can greatly benefit from reusing results from previous queries, as simply caching intermediate results is a lot less costly than performing all calculations.
The implementation of the result hash table already covers the first aspect: The results from the `evaluation` CTE are written into a hash table.
To be able to use the stored results in the hash table, the second aspect has to be implemented. This can be done almost the same way as in the original `dtw` memoization version, where the results are queried from the memoization table inside CTE `call_graph`. Instead of the memoization table, we call the hash table at this point.
The hash tables are stored in the working memory. It means that the memoization using hash tables is only available per session. In contrast the original version, that uses a table, stores the data in persistent memory making the memoization available across sessions.

In the next section we will explain how memoization can be activated using the result hash table. Here only the concepts are explained. The complete SQL code implementation can be found in the appendix 6 at the end of this thesis.

In order for the hash table to exist outside of the function call, we need to create it before the function is called. So we remove `prepareHT` of the result hash table in the function body and move it before the function call.
Since the first step of writing the results to the table is no longer necessary, the additional CTE `memoization` at the end of the UDF, which is used for storing, can be removed. The results are written into the hash table directly in the evaluation and then used to read the final result. In following function calls in the same session these can be used as base cases.
In the `call_graph` CTE the values from the memoization table are read inside the `memoization` CTE. We replace the access on the memoization table with a lookup on the result hash table.

The result of the modifications is a `dtw` memoization version that benefits from two hash

tables and additionally stores results for later invocations in the working memory.

In the following, we focus on the SQL plan changes between the `dtw` memoization version and its hash table version. Then we look at the impact on the runtimes.

4.4.1 SQL plan change

The `dtw` memoization version is based on the reference version and additionally implements memoization. Therefore, the SQL plans of the two versions match to a large extent. There are only two differences between the plans:
The memoization version has an additional CTE `memoization` within the CTE `call_graph`, which is responsible for calling the values from the memoization table.
At the end of the UDF another CTE called `memoization` was added, which stores the contents of the `evaluation` CTE into the memoization table.

Therefore the plan changes described in chapter 4.3.1 are also part of the plan changes in this section. In addition, there are two other plan changes in the memoization parts. The first one takes place in the CTE `memoization`, that is included in the `call_graph`, see figures 4.29 and 4.30. Instead of a Index Scan of the memoization table in the persistent storage, a hash table lookup is performed in the form of a Function Scan.
The second change is the removal of the storage CTE `memoization`.

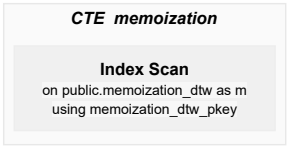


Figure 4.29: SQL plan of the CTE `memoization`, that is inside the CTE `call_graph`, of the `dtw` memoization version without hash tables.

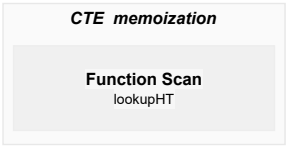


Figure 4.30: SQL plan of the CTE `memoization`, that is inside the CTE `call_graph`, of the `dtw` memoization version with call graph and result hash table.

In the next section, we look at the runtimes of the `dtw` memoization version.

4.4.2 Runtime results

In this section, first we consider which impact the storing of results has on the runtime. In this case we do not use the beneficial effect of memoization, which uses the stored results in subsequent function executions. We intentionally discard the stored results of previous executions before another execution takes place.
In the next part we activate the memoization and see how the new hash table version performs compared to the original version. And if the memoization still works as expected.

Comparison without using memoization

As before, the runtime measurements are created under the same conditions as in chapter 4.1.2.
In addition, the advantage of memoization is purposefully suppressed. After each function call, the stored results are intentionally discarded. This allows to consider how high the price of storing the results for subsequent executions is.
In figure 4.31 you can see that adding memoization generally increases the runtime slightly. The versions without memoization were already shown in previous chapters. The `dtw` reference version with both hash tables still has the best performance. The memoization version described in this chapter is slightly slower. It uses the result hash table to store the results.

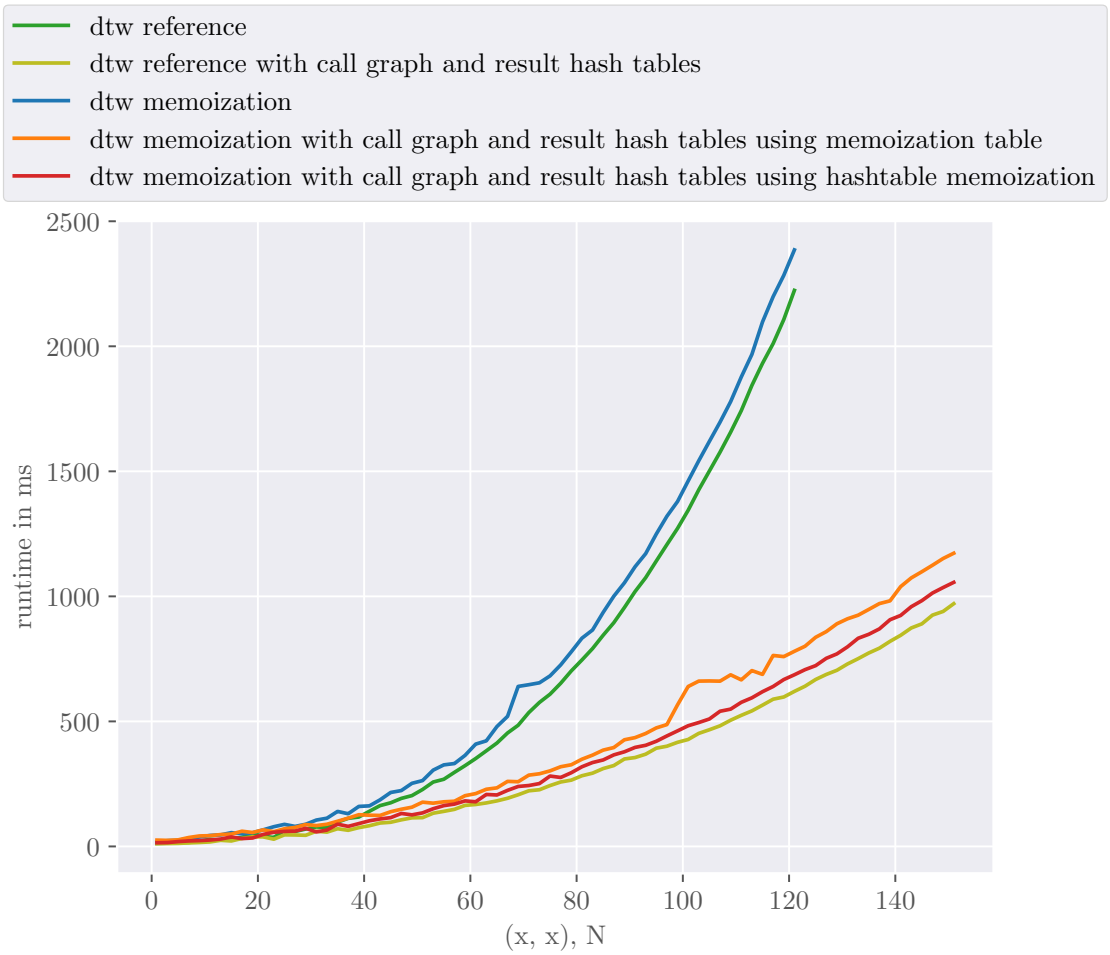


Figure 4.31: Comparison of different `dtw` versions with and without save of results in a memoization table or in a hash table.

In addition, the runtimes of a previously not described version were measured: `dtw` memoization with both hash tables, which still uses the original variant of memoization. This version is slightly slower than the hash table memoization version.
The benefit of memoization is very large as we will see in the following chapter. Using a result from previous executions can greatly reduce the runtime, as already shown in the paper of Duta and Grust [12].

Memoization effect

In the following measurements the advantage of memoization is used. This means storing the intermediate results, and using them in subsequent function calls.

The experiment was created by running 100 function calls on a random list of input values in range from (1, 1) to (100, 100). All versions use the same randomly generated list. The results from earlier executions calls are accessible to later executions calls. This means a function that uses the hash table for memoization needs to be called in a session where all previous calls of the random list were executed.

Figure 4.32 shows the comparison between the different `dtw` memoization versions. For comparison, a version is included where the stored results are not used.
All versions benefit strongly from memoization. The versions that use hash tables generally perform slightly better. Using the hash table for memoization results in an even lower runtime, when results are present from earlier function executions.
When using the memoization table for storing, the results are written into the persistent memory. This has an impact on the runtime, since reading from persistent memory takes longer than reading them from working memory in case of the hash table.
The persistent table has the advantage of being available across multiple sessions. However, this can also be a disadvantage if, for example, different sessions are working on different data sets. In this case the hash table is beneficial because it makes the stored values available only within the session.

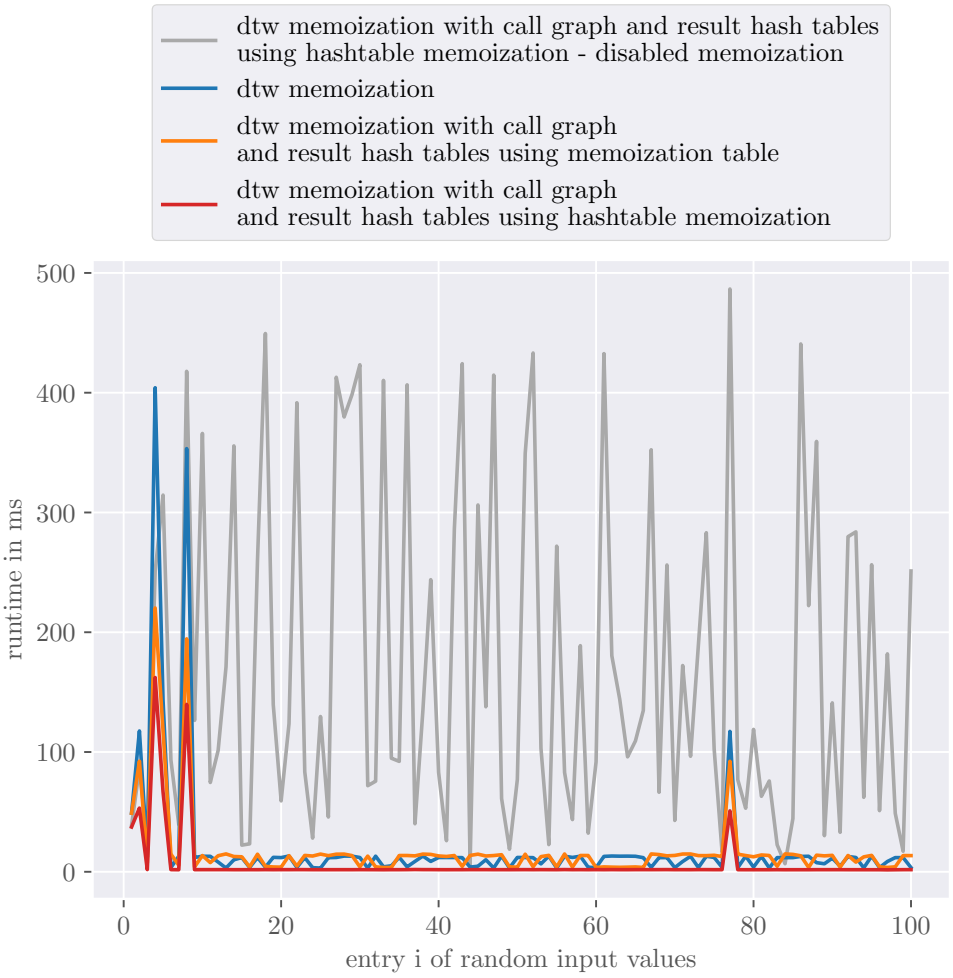


Figure 4.32: `dtw` memoization using memoization effect to evaluate 100 random values in range from (1,1) to (100, 100). Initial function executions must calculate the results themselves and therefore have a high execution time. Later function executions can use stored results, which results in a significantly lower execution time. That function which does not use memoization cannot benefit from stored values and has throughout high execution times.

5

Application of hash tables to other fsUDF algorithms

In chapter 4, we discussed in detail the implementation of the call graph and the result hash tables on the example of dynamic time warping. In this chapter we focus on other algorithm examples.

For each example, a reference function is created using the fsUDF template [12]. This function is equivalent to the considered `dtw` function in chapter 4.3. We use the reference version, since it is the template with the best performance.
In the following subchapters, we will compare the following four versions in each case:

- without hash tables
- with call graph hash table
- with result hash table
- with call graph and result hash tables

The reference template results in equivalent function versions for each example, where the two hash tables can be integrated the same way like in chapter 4.3. Only the number of parameters differ. This means that for each example the same SQL plan changes can be observed.

For the runtime experiments, we use a similar setup as in the previous chapter 4, but we leave the default value for the PostgreSQL configuration parameter `random_page_cost` [6], because we have no need to enforce a preference of index scans over sequential scans.

5.1 Graph reachability algorithm

The graph reachability algorithm tests if two vertices are connected in a directed acyclic graph (DAG) and has a 2-fold recursion. We call it `comps` like in the paper of Duta and Grust [12].

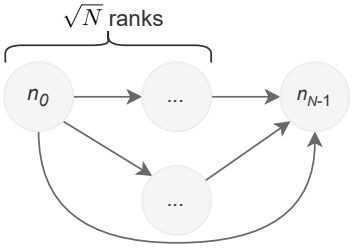


Figure 5.1: structure of DAG $G = (V, E)$

For the runtime evaluation the algorithm runs on a DAG $G = (V, E)$ with $V = \{n_0, ..., n_{N-1}\}$ and $|E| = N - 1$, see figure 5.1. The vertex n_0 is always the start vertex and n_{N-1} always the end point of the algorithm. A vertex n_i where $0 < i < N$ is on one of $\sqrt{N}$ ranks. There are only edges from vertices on lower to vertices on higher ranks.

The runtime results are shown in figure 5.2. We can see, that the version using the call graph hash table has almost the same runtime as the basic version without hash tables. The versions with the result hash table perform better.
It means only the usage of the result hash table increases the performance. This can be explained with the table sizes, which we can see in figure 5.3.
The overall size of the call graph is relatively small compared to the evaluation table. This is why a hash table for the call graph has no big impact on the runtime. Accessing specific entries in the call graph is always fast, even when using sequential CTE scans. In contrast the `evaluation` CTE is huge and therefore a hash table brings a big performance boost.

In total for the algorithm `comps` the usage of hash tables only leads to a slightly increased performance, where only the result hash table has an impact on the runtime.

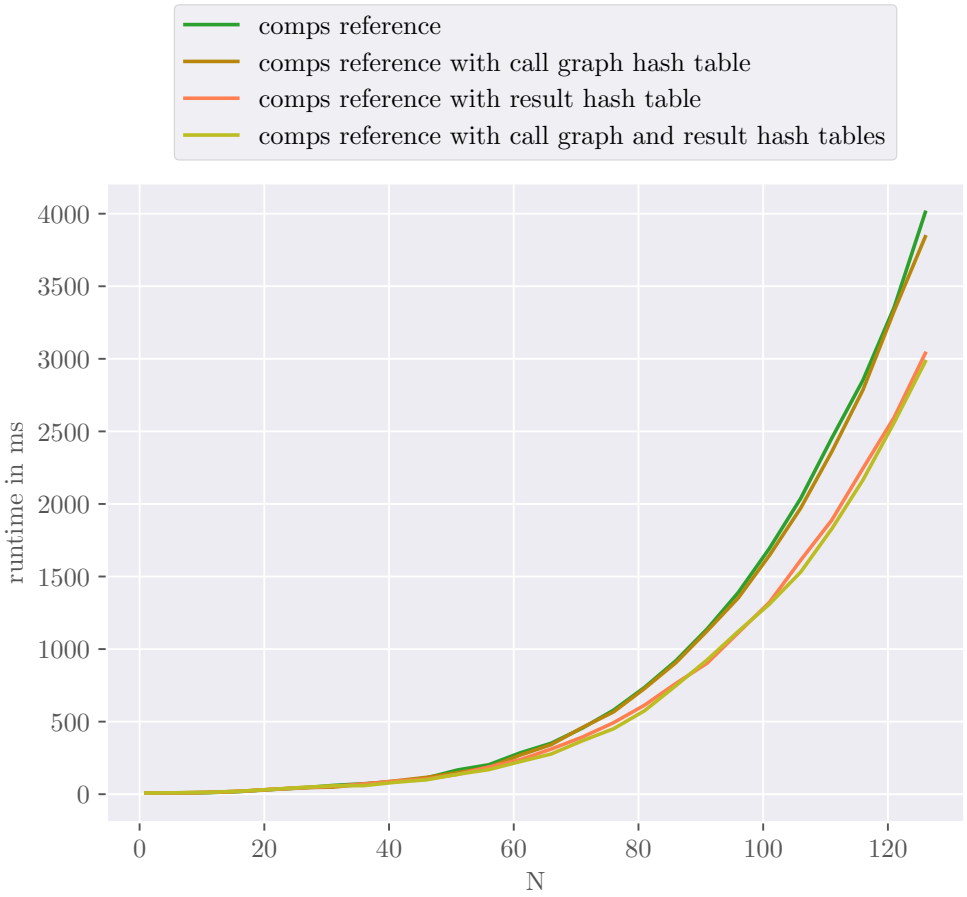


Figure 5.2: Runtime results of algorithm `comps` using the reference template with and without call graph and result hash tables

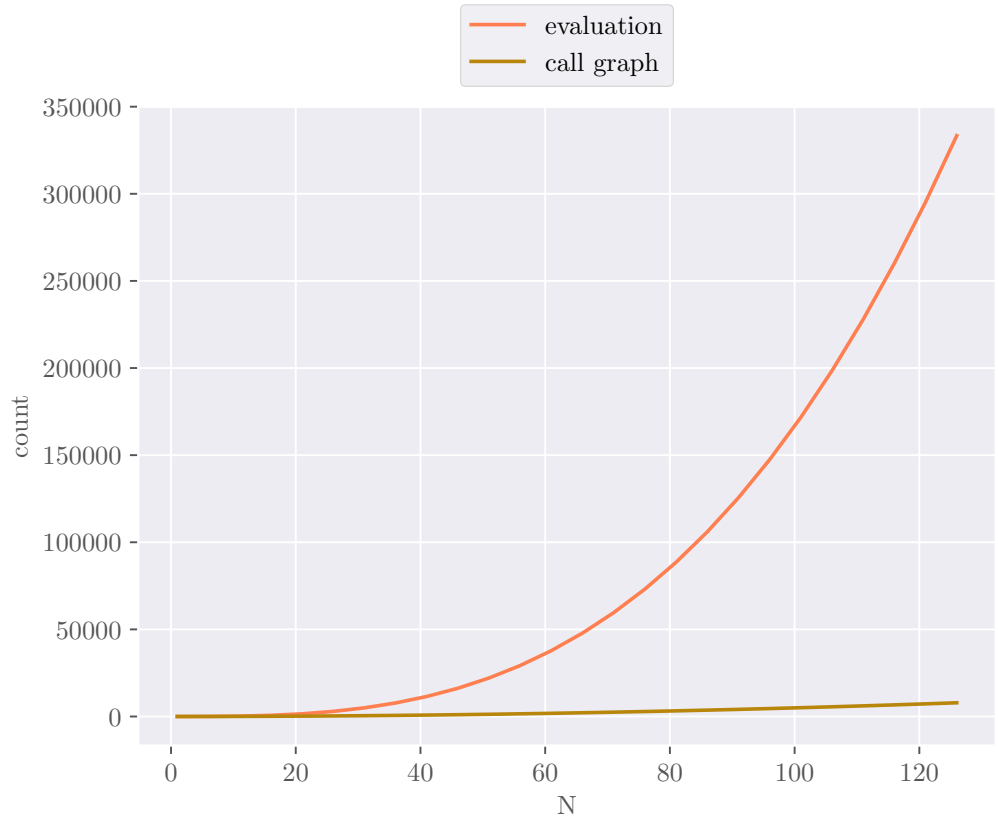


Figure 5.3: Table sizes of the `call_graph` and `evaluation` CTE in the `comps` reference fsUDF

5.2 Longest common path algorithm

The algorithm longest common path (`lcp`) determines the length of the longest common path with matching labels in a DAG starting at any two given vertices and has a 2-fold recursion.
In figure 5.4 the graph structure is shown, which we use for the runtime experiment. The graph $G = (V, E)$ has $|V| = N$ vertices. Each of them is ordered into $\sqrt{N}$ ranks, where n_0 is in rank 0. The vertex n_0 is used as the start point of both vertices for the algorithm. So that the of the longest common path is always $\sqrt{N}$.
Only edges from vertices on lower ranks to vertices on higher ranks exist. An edge exists between any two vertices with a probability of 30%, except for vertices whose ranks differ by exactly 1, for these there is always an edge.

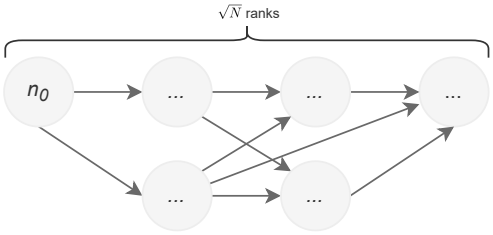


Figure 5.4: structure of DAG $G = (V, E)$

The result of the run time measurements (see figure 5.5) shows that only the result hash table has an impact on the performance of the function. The call graph hash table does not improve the runtime, but it does not worsen it either.

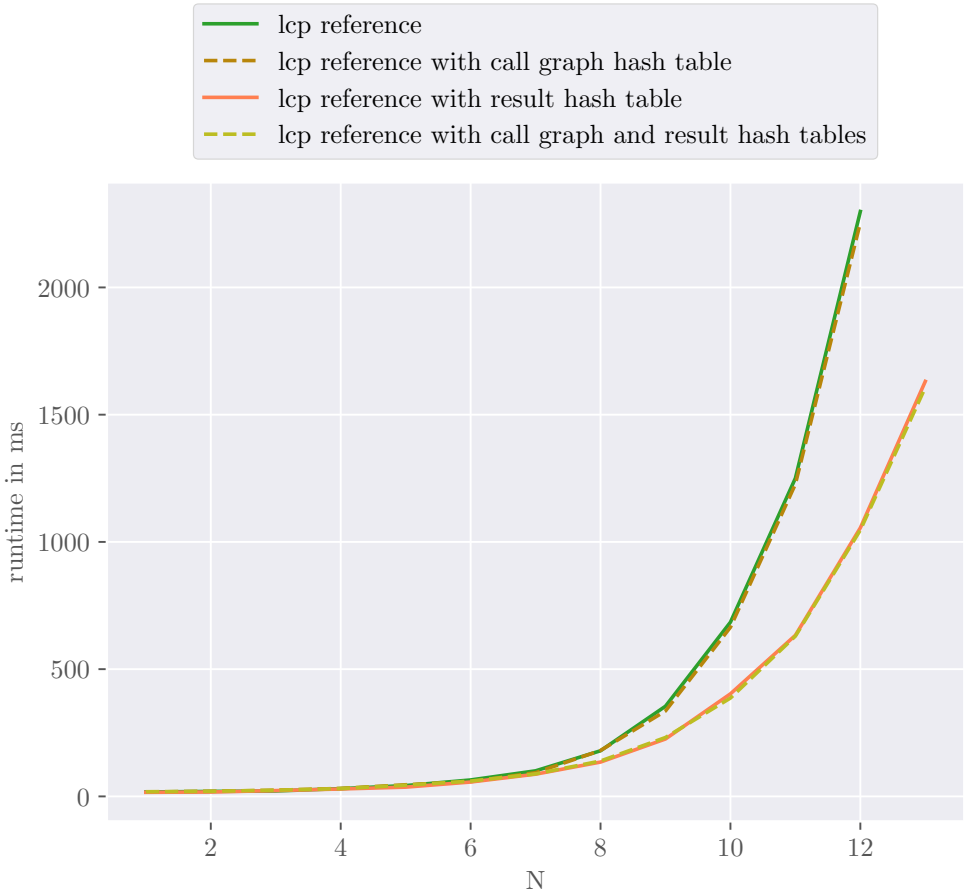


Figure 5.5: Runtime results of algorithm `lcp` using the reference template with and without call graph and result hash tables

The result hash table improves the performance of the function. Similar to the `dtw` example, the performance improvement is exponentially increasing.

In figure 5.6 we can see table sizes of the CTE `evaluation` and `call_graph`. The call graph size is small, which means CTE scans can return desired rows relatively fast. This means that no simulated index is necessary on them. The table size of the CTE `evaluation` is significantly larger, which makes it worth using the hash table, since the runtime of sequential scans increases significantly with large table sizes. From this we can conclude why only the result hash table has a positive effect on the runtime.
The results of this is example is very similar to the example with the algorithm `comps` (see chapter 5.1). The call graph hash table has no or little impact on the runtime and the call graph size is small.

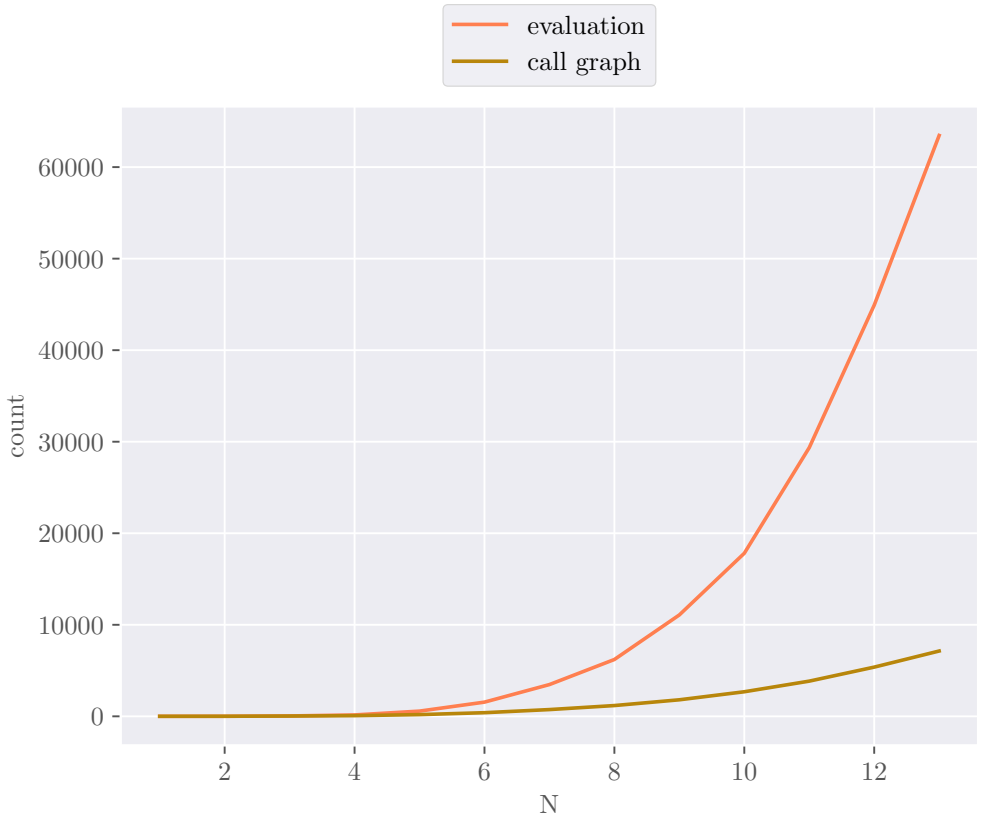


Figure 5.6: Table sizes of the `call_graph` and `evaluation` CTE in the `lcp` reference fsUDF

5.3 Longest common subsequence algorithm

The `lcs` algorithm finds the longest common subsequence of two strings and has a 2-fold recursion.
As inputs for the runtime evaluation we generate two random strings with length N.

In figure 5.7 we can see the runtime results. The `lcs` reference version benefits from both available hash tables. Both versions with hash tables perform better compared to the original version. The best performance has the version, that uses both hash tables. The benefit of the result hash table is higher than the benefit of the call graph hash table.

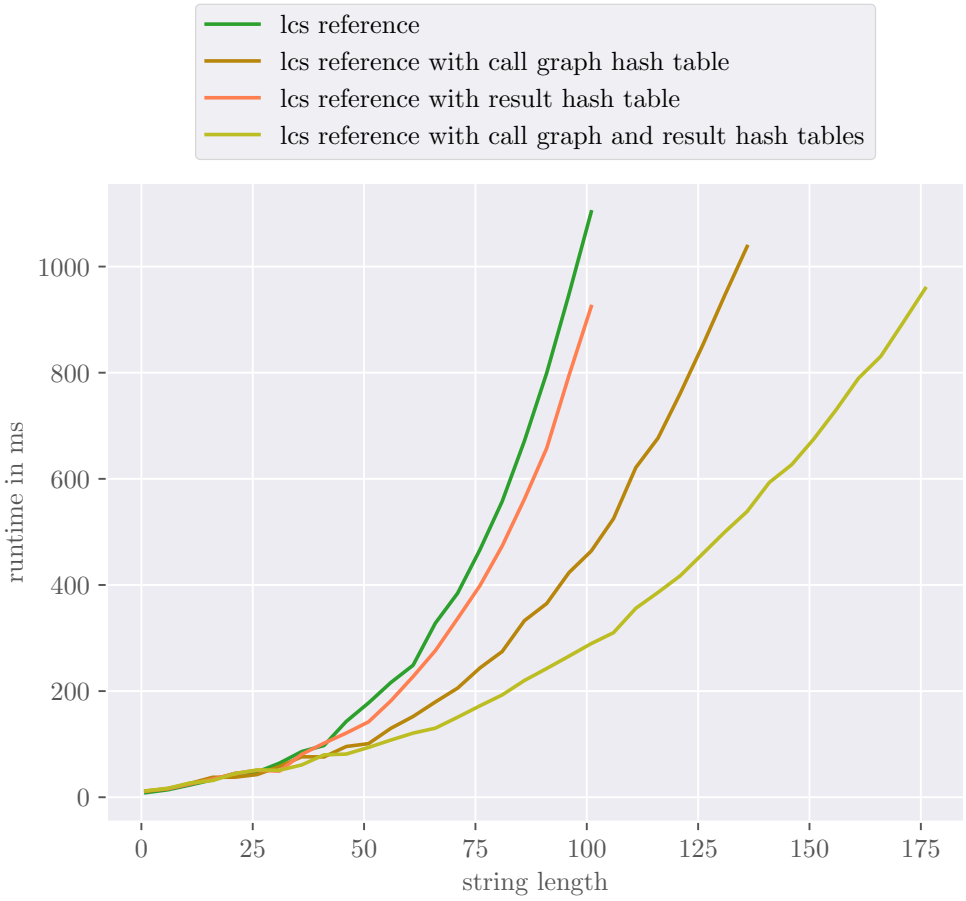


Figure 5.7: Runtime results of algorithm `lcs` using the reference template with and without call graph and result hash tables

In figure 5.8 we can see the table sizes of the `call_graph` and `evaluation` CTE tables. In the case of the `lcs` algorithm the evaluation table is larger than the call graph table. We observe, that the result hash table has a smaller performance boost, although its

table is larger.
A possible explanation is that the evaluation table is already used for computation when it hasn't reached its full size. In addition `lcs` is a 2-fold recursion, which is why there are less calls on the result hash table compared to other examples and therefore less beneficial.

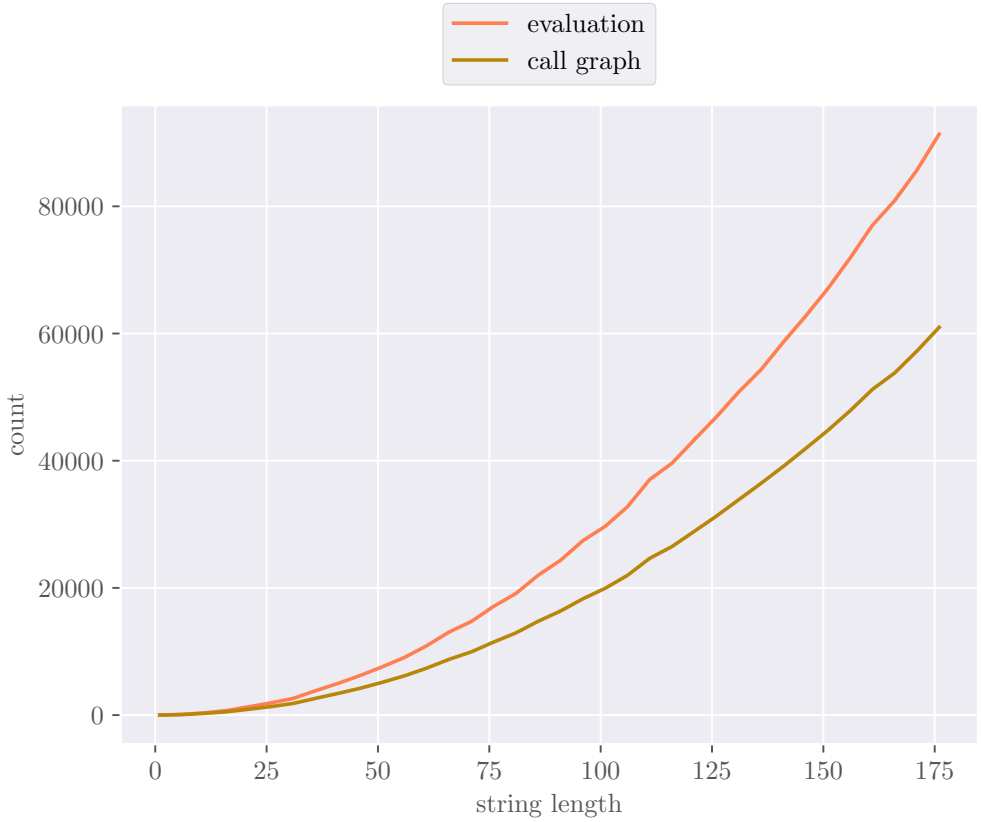


Figure 5.8: Table sizes of `call_graph` and `evaluation` CTE in the `lcs` reference fsUDF

5.4 Floyd-Warshall algorithm

The Floyd-Warshall algorithm or short `floyd` determines the length of the shortest path in a directed graph and has a 3-fold recursion. [11]
The graph $G = (V, E)$ on which the `floyd` algorithm runs in this experiment is random and has $|V| = N$. Each edge e has a weight value, which is randomly generated. There is a 10% probability that an edge exists between a pair of vertices. The start and end vertex of each execution are two selected vertices in the graph.
In figure 5.9 we observe that the call graph hash table only gives a little performance

boost. The result hash table has a much larger positive impact on the performance. The version using both hash tables has the best performance.

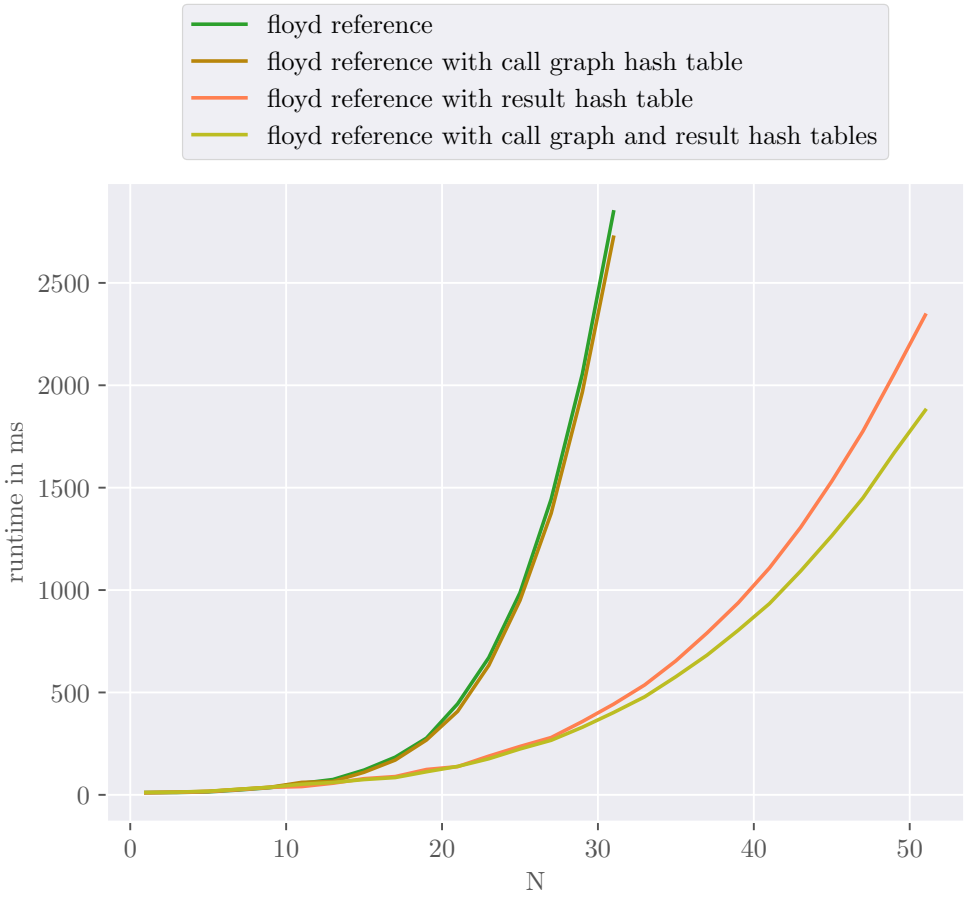


Figure 5.9: Runtime results of algorithm `floyd` using the reference template with and without call graph and result hash tables

As the table sizes are almost identical the reason for the performance difference is not clear, see figure 5.10. Due to the 3-fold recursion of the algorithm the result hash table is more frequently used, which can lead to a larger performance boost. It is not clear, why the call graph hash table has almost no positive impact on the performance. In case of `dtw`, which has also a 3-fold recursion, the call graph table is smaller, but still brings a performance enhancement.

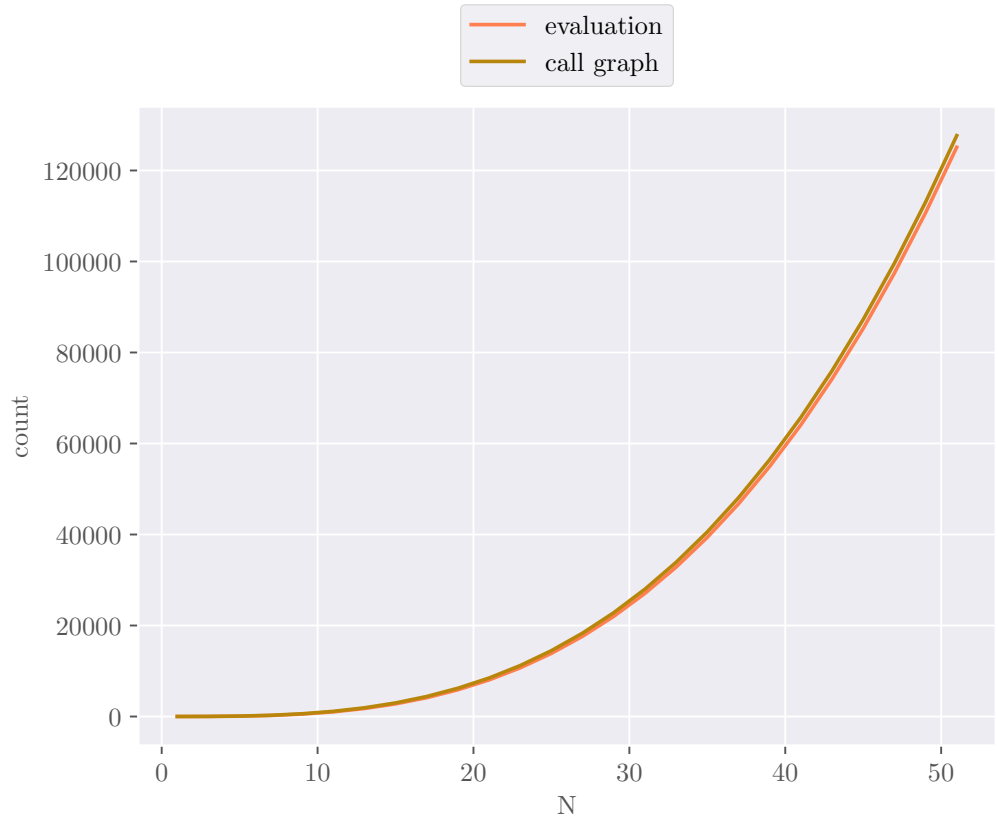


Figure 5.10: Table sizes of the `call_graph` and `evaluation` CTE in the `floyd` reference fsUDF

5.5 Finite state machine algorithm

The finite state machine algorithm (`fsm`) parses molecule names using a state machine. It is the only linear recursion algorithm considered.
As evaluation data we use infinite extendable molecule name with length N.
In figure 5.11 we can observe a significant performance boost. The versions that use the call graph hash table have a almost linear running time curve instead of a exponential one. Using the call graph hash table has a huge impact on the runtime. The result hash table has no observable performance benefit on the runtime of the function.

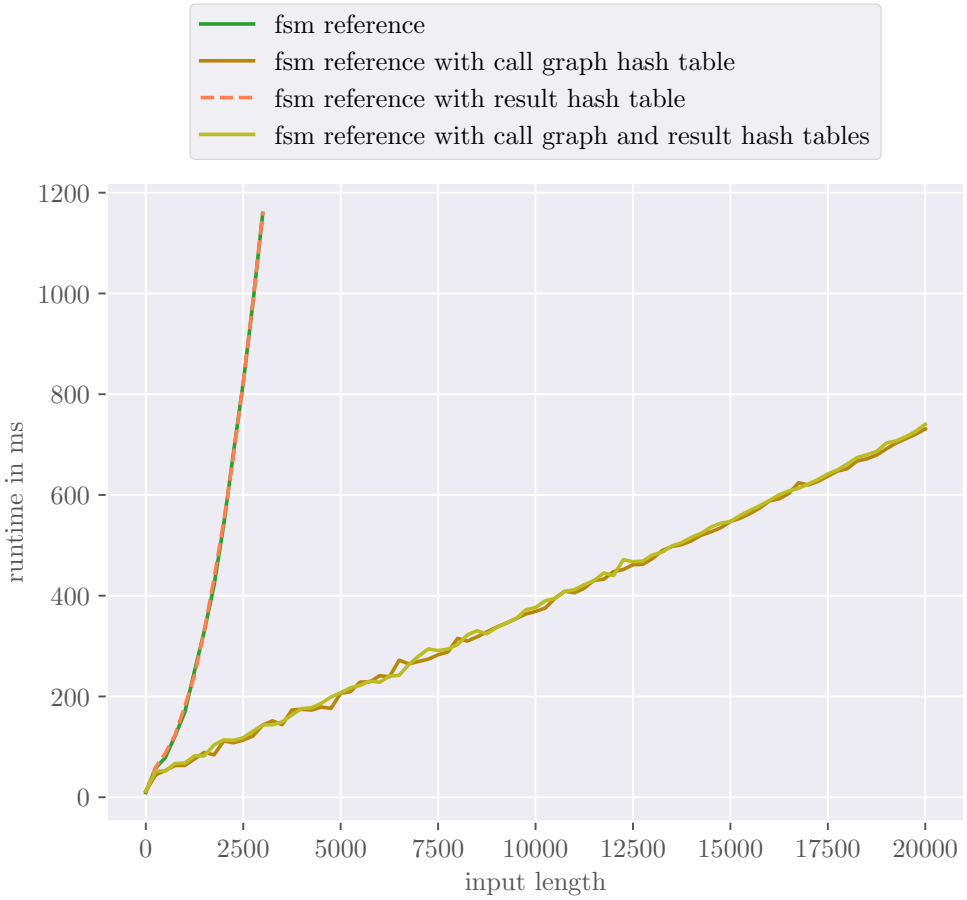


Figure 5.11: Runtime results of algorithm `fsm` using the reference template with and without call graph and result hash tables

The table sizes are identical, which can be explained by the linear recursion of the algorithm. The algorithm has a linear recursion, which means there are less calls on the result hash table. This can serve as a possible explanation why the impact of the result hash table is not observable. The usage of the result hash table does not worsen the

runtime performance either.

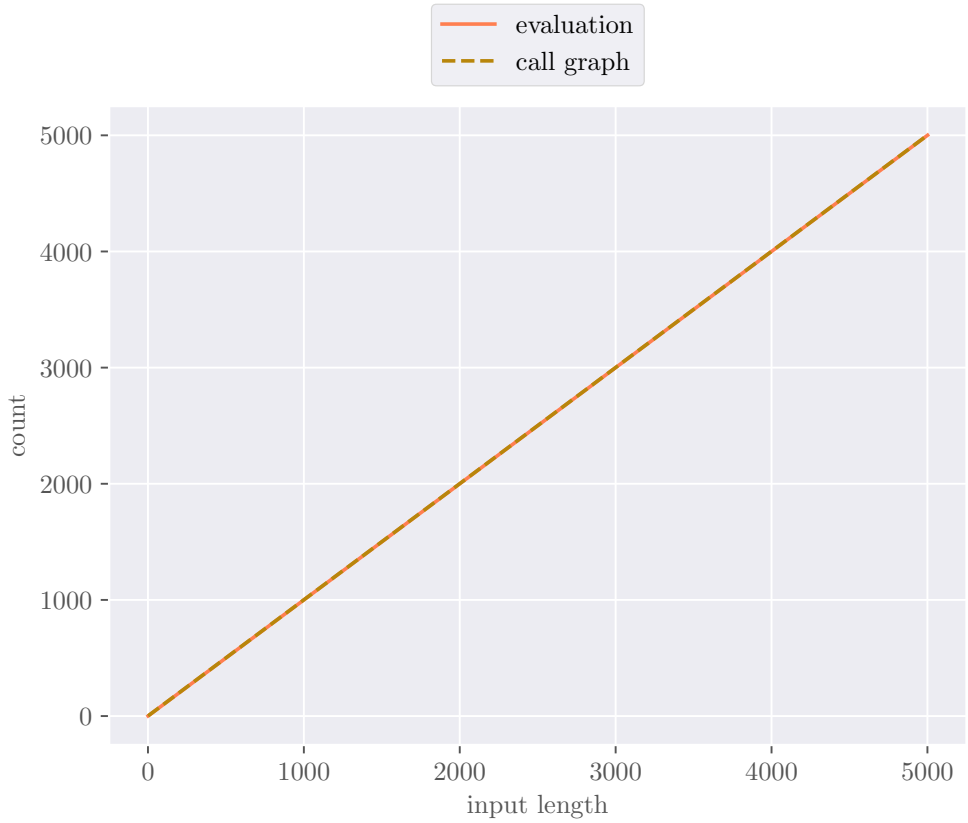


Figure 5.12: Table sizes of the `call_graph` and `evaluation` CTE in the `fsm` reference fsUDF

6

Conclusion

In this thesis, a proof of concept of CTE performance improvements using a PostgreSQL hash table extension was realized. Based on the example of the functional-style UDFs of Duta and Grust [12], replacing costly CTE scans on the CTEs `call_graph` and `evaluation` with significantly faster hash table lookups, led to performance boosts for all considered algorithms. Accordingly, the goal of improving the performance of the UDF with the help of hash tables has been achieved.
Especially when the CTE on which the hash table based simulated index is applied is large, a significant performance increase can be observed. For small CTE tables, no performance improvement can be observed, but no performance degradation either. In case of the `fsm` algorithm example the usage of the call graph hash table led from a exponential to a almost linear runtime, which allows operating on vastly bigger inputs in an acceptable time.
Using hash tables in UDFs can increase the performance immensely, but there are still open issues. The current implementation of this proof of concept is not ideal, because of the enforcement of a specific execution order to guarantee a correct execution of the function. This is not optimal as special care needs to be taken to produce correct results, which is not user friendly. Further improvements that make the procedure more seamless would greatly enhance this approach.
Nevertheless, this approach of using hash tables to improve the performance of CTEs and also UDFs is very promising. A mature variant of the index could be a PostgreSQL feature, which creates an index for a CTE by using a simple flag to increase its performance.

Following future work can be proposed to improve this concept:

- Implementation of the option of a CTE index.
  It could be implemented using automatically generated hash tables that transfer the full table contents of the CTE into a hash table. Using the hash table, some CTE scans can be replaced with hash table lookups to increase the performance.

- Adding the support of storing multiple values on the same key to the hash table extension.
  As a result, tables with keys which are not unique no longer need to use an array of a composite type to store multiple values on the same key. This could eliminate the necessary `unnest` function call that was previously needed to access the values in the array.
- Implementation of hash tables, that do not need to enforce a certain execution order. Before lookup statements can be executed on the hash table, all insert statements from previous CTEs must have been executed.
- Application of hash tables on other UDF examples with CTEs.
- Creation of a functional-style UDF template, that uses hash tables. The hash table template can be based on the reference or memoization template and would enable simple conversions from common algorithms to hash table optimized variants.

Bibliography

[1] Build tuple hash table ext. https://doxygen.postgresql.org/execGrouping_8c_source.html#l00154. Accessed: 03-04-2022.

[2] Chapter 11. indexes. https://www.postgresql.org/docs/13/indexes.html. Accessed: 28-03-2022.

[3] hash table. https://xlinux.nist.gov/dads/HTML/hashtab.html. Accessed: 06-04-2022.

[4] Index types. https://www.postgresql.org/docs/13/indexes-types.html. Accessed: 06-04-2022.

[5] Lookup tuple hash entry. https://doxygen.postgresql.org/execGrouping_8c_source.html#l00306. Accessed: 03-04-2022.

[6] Query planning. https://www.postgresql.org/docs/13/runtime-config-query.html. Accessed: 27-03-2022.

[7] Reference counting. https://www.educative.io/courses/a-quick-primer-on-garbage-collection-algorithms/jR8ml. Accessed: 03-05-2022.

[8] Tuple hash tables. https://doxygen.postgresql.org/execnodes_8h_source.html#l00736. Accessed: 03-04-2022.

[9] With queries (common table expressions). https://www.postgresql.org/docs/13/queries-with.html. Accessed: 28-03-2022.

[10] *Dynamic Time Warping*, pages 69–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[12] C. Duta and T. Grust. Functional-style SQL UDFs with a capital 'F'. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1273–1287, New York, NY, USA, 2020. Association for Computing Machinery.

[13] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[14] M. S. Miller, D. von Dincklage, V. Ercegovac, and B. Chin. Uncanny valleys in declarative language design. In *SNAPL 2017, Summit on Advances in Programming Languages*, 2017.

Appendix

dtw data preparation

```
DROP TABLE IF EXISTS X;
CREATE TABLE X
(
    t serial PRIMARY KEY,
    x DOUBLE PRECISION
);

DROP TABLE IF EXISTS Y;
CREATE TABLE Y
(
    t serial PRIMARY KEY,
    y DOUBLE PRECISION
);

SELECT setseed(0.42);

INSERT INTO X(t, x)
VALUES (1, 0),
       (2, 0),
       (3, 1),
       (4, 2),
       (5, 0);

INSERT INTO X(t, x)
SELECT t, random() AS x
FROM generate_series(6, :N) AS _(t);

INSERT INTO Y(t, y)
VALUES (1, 0),
       (2, 1),
       (3, 1),
       (4, 1),
       (5, 0);
```

```
34
35  INSERT INTO Y(t, y)
36  SELECT t, random() AS y
37  FROM generate_series(6, :N) AS _(t);
38
39  ANALYZE X;
40  ANALYZE Y;
```

Figure 1: Preparation of the data tables on which the dtw function can be executed

dtw default

```
1   CREATE FUNCTION f(args)
2   RETURNS DOUBLE PRECISION
3   AS $$
4   WITH RECURSIVE
5       call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
6       (See figure 3),
7       base_cases(in_i, in_j, val) AS
8       (See figure 4),
9       evaluation(in_i, in_j, val) AS
10      (See figure 5))
11  SELECT  e.val
12  FROM    evaluation AS e
13  WHERE   (e.in_i, in_j) = ((dtw.args).i, (dtw.args).j);
14  $$ LANGUAGE SQL STABLE
15                  STRICT;
```

Figure 2: Function body of dtw default

```
1   call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS (
2       SELECT (dtw.args).i,
3              (dtw.args).j,
4              NULL :: INT,
5              NULL :: BIGINT,
6              (dtw.args).i,
7              (dtw.args).j,
8              NULL :: DOUBLE PRECISION
9
10      UNION
11
12      SELECT g.out_i, out_j, edges.*
13      FROM call_graph AS g,
14       LATERAL (
15           WITH slices(site, out_i, out_j) AS (
16               SELECT 1,
17                      (
18                      SELECT  *
19                      FROM    (SELECT (NULL, false)::lifted_args) AS _
20                      WHERE   (g.out_i = 0 AND g.out_j = 0)
21                      UNION ALL
```

```
22                      SELECT  *
23                      FROM    (SELECT (NULL, false)::lifted_args) AS _
24                      WHERE   NOT (g.out_i = 0 AND g.out_j = 0)
25                              AND (g.out_i = 0 OR g.out_j = 0)
26                      UNION ALL
27                      SELECT  *
28                      FROM    (SELECT (ROW  (g.out_i - 1, g.out_j - 1),
29                                            true)::lifted_args
30                              FROM (X JOIN Y ON
31                                   ((X.t, Y.t) = (g.out_i, g.out_j)))
32                                   AS Z) AS _
33                      WHERE   NOT ((g.out_i = 0 AND g.out_j = 0)
34                              OR (g.out_i = 0 OR g.out_j = 0))
35              )
36      UNION ALL
37      SELECT 2,
38              (
39              SELECT  *
40              FROM    (SELECT (NULL, false)::lifted_args) AS _
41              WHERE   (g.out_i = 0 AND g.out_j = 0)
42              UNION ALL
43              SELECT  *
44              FROM    (SELECT (NULL, false)::lifted_args) AS _
45              WHERE   NOT (g.out_i = 0 AND g.out_j = 0)
46                AND   (g.out_i = 0 OR g.out_j = 0)
47              UNION ALL
48              SELECT  *
49              FROM    (SELECT (ROW  (g.out_i - 1, g.out_j),
50                                    true)::lifted_args
51                      FROM (X JOIN Y ON
52                           ((X.t, Y.t) = (g.out_i, g.out_j)))
53                           AS Z) AS _
54              WHERE   NOT ((g.out_i = 0 AND g.out_j = 0)
55                      OR (g.out_i = 0 OR g.out_j = 0))
56              )
57      UNION ALL
58      SELECT 3,
59              (
60              SELECT  *
61              FROM    (SELECT (NULL, false)::lifted_args) AS _
62              WHERE   (g.out_i = 0 AND g.out_j = 0)
63              UNION ALL
64              SELECT  *
65              FROM    (SELECT (NULL, false)::lifted_args) AS _
66              WHERE   NOT (g.out_i = 0 AND g.out_j = 0)
67                AND   (g.out_i = 0 OR g.out_j = 0)
68              UNION ALL
69              SELECT  *
70              FROM    (SELECT (ROW  (g.out_i, g.out_j - 1),
71                                    true)::lifted_args
72                      FROM (X JOIN Y ON
73                           ((X.t, Y.t) = (g out_i, g.out_j)))
74                           AS Z) AS _
```

```
75                      WHERE   NOT ((g.out_i = 0 AND g.out_j = 0)
76                              OR (g.out_i = 0 OR g.out_j = 0))
77                      )
78          ),
79               calls(site, fanout, i, j, val) AS (
80                   SELECT    s.site,
81                             COUNT(*) OVER (),
82                             (s.out).args.i,
83                             (s.out).args.j,
84                             NULL :: DOUBLE PRECISION
85                   FROM      slices AS s
86                   WHERE     (s.out).not_bottom
87               )
88          TABLE calls
89
90          UNION ALL
91
92          SELECT NULL :: INT,
93                 0,
94                 g.out_i, g.out_j,
95                 (
96                     SELECT CASE
97                                WHEN g.out_i = 0 AND g.out_j = 0
98                                THEN 0 :: DOUBLE PRECISION
99                                WHEN g.out_i = 0 OR g.out_j = 0
100                               THEN 'infinity' :: DOUBLE PRECISION
101                               ELSE (SELECT abs(Z.x - Z.y) +
102                                            LEAST(NULL::DOUBLE PRECISION,
103                                                  NULL::DOUBLE PRECISION,
104                                                  NULL::DOUBLE PRECISION)
105                                     FROM (X JOIN Y ON
106                                          ((X.t, Y.t)
107                                          = (g.out_i, g.out_j))
108                                          ) AS Z)
109                               END)
110         WHERE NOT EXISTS(TABLE calls)
111         ) AS edges(site, fanout, i, j, val)
112 ),
```

Figure 3: CTE `call_graph` used in dtw default and its call graph hash table version

```
1  base_cases(in_i, in_j, val) AS (
2      SELECT g.in_i, g.in_j, g.val
3      FROM call_graph AS g
4      WHERE g.fanout = 0
5  ),
```

Figure 4: CTE `base_cases`, used in dtw default

```
1  evaluation(in_i, in_j, val) AS (
2      TABLE base_cases
3      UNION ALL
4      (
5          WITH e AS (TABLE evaluation),
6               returns(in_i, in_j, val) AS (
7                   SELECT go.in_i,
8                          go.in_j,
9                          (
10                         SELECT
11                             CASE
12                             WHEN go.in_i = 0 AND go.in_j = 0
13                             THEN 0 :: DOUBLE PRECISION
14                             WHEN go.in_i = 0 OR go.in_j = 0
15                             THEN 'infinity' :: DOUBLE PRECISION
16                             ELSE (SELECT  abs(Z.x - Z.y)
17                                           + LEAST(  go.ret[1],
18                                                     go.ret[2],
19                                                     go.ret[3])
20                                   FROM    (X JOIN Y ON
21                                            ((X.t, Y.t)
22                                            = (go.in_i, go.in_j)))
23                                           AS Z)
24                                  END)
25                  FROM (SELECT   g.in_i,
26                                 g.in_j,
27                                 array_gather(e.val, g.site) AS ret
28                        FROM     call_graph AS g,
29                                 e
30                        WHERE    (g.out_i, g.out_j) = (e.in_i, e.in_j)
31                          AND    NOT EXISTS(SELECT
32                                            FROM    e
33                                            WHERE  (e.in_i, e.in_j)
34                                                 = (g.in_i, g.in_j))
35                        GROUP BY (g.in_i, g.in_j), g.fanout
36                        HAVING COUNT(*) = g.fanout
37                       ) AS go(in_i, in_j, ret)
38              )
39          SELECT results.*
40          FROM (TABLE e UNION ALL TABLE returns) AS results
41          WHERE NOT EXISTS( SELECT
42                            FROM    e
43                            WHERE (e.in_i, e.in_j)
44                                = ((dtw.args).i, (dtw.args).j))
45      ))
```

Figure 5: CTE `evaluation`, used in dtw default and its call graph hash table version

dtw default with call graph hash table

```
1  CREATE TYPE CALLGRAPHVAL AS
2  (
3      in_i     INTEGER,
4      in_j     INTEGER,
5      site     INTEGER,
6      fanout   INTEGER,
7      val      DOUBLE PRECISION
8  );
```

Figure 6: Creation of composite type **CALLGRAPHVAL**

```
1  CREATE FUNCTION f(args)
2  RETURNS DOUBLE PRECISION
3  AS $$
4   SELECT prepareHT(CG_HT_ID, 2,     NULL::INT,
5                                     NULL::INT,
6                                     NULL::CALLGRAPHVAL[]);
7  WITH RECURSIVE
8      call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
9      (See figure 3),
10     fill_hashtable AS
11     (See figure 8),
12     base_cases(in_i, in_j, val) AS
13     (See figure 9),
14     evaluation(in_i, in_j, val) AS
15     (See figure 5 with replaced returns CTE in figure 10)
16 SELECT  e.val
17 FROM    evaluation AS e
18 WHERE   (e.in_i, e.in_j) = ((dtw.args).i, (dtw.args).j);
19 $$ LANGUAGE SQL STABLE
20                 STRICT;
```

Figure 7: Function body of dtw default with call graph hash table. Red areas mark the differences compared to the original version

```
1  fill_hashtable AS (
2      SELECT  g.out_i, g.out_j,
3              insertToHt(CG_HT_ID, true, g.out_i, g.out_j,
4                         array_agg((g.in_i, g.in_j, g.site, g.fanout, g.val)
5                          :: CALLGRAPHVAL) :: CALLGRAPHVAL[])
6      FROM    call_graph as g
7      GROUP BY g.out
8  ),
```

Figure 8: CTE `fill_hashtable`, used in all versions, that use the call graph hash table

```
base_cases(in_i, in_j, val) AS (
    SELECT  g.in_i, g.in_j, g.val
    FROM    call_graph AS g,
            (SELECT COUNT(*) FROM fill_hashtable) AS _
    WHERE   g.fanout = 0
),
```

Figure 9: CTE `base_cases` with usage of the call graph hash table. Used in dtw default with call graph hash table

```
returns(in_i, in_j, val) AS (
 SELECT go.in_i,
        go.in_j
        (SELECT
            CASE
                WHEN go.in_i = 0 AND go.in_j = 0
                THEN 0.0
                WHEN go.in_i = 0 OR go.in_j = 0
                THEN 'infinity' :: DOUBLE PRECISION
                ELSE    (SELECT abs(Z.x - Z.y)
                                + LEAST(go.ret[1], go.ret[2], go.ret[3])
                         FROM   (X JOIN Y ON
                                (X.t, Y.t) = (go.in_i, go.in_j))) AS Z)
            END)
 FROM (SELECT   g.in_i, g.in_j, array_gather(e.val, g.site) AS ret
       FROM     e,
                lookUpHT(CG_HT_ID, false, e.in_i, e.in_j)
                    AS ht(out_i INT, out_j INT, cgval CALLGRAPHVAL[]),
                unnest(ht.cgval) AS g
       WHERE    NOT EXISTS( SELECT
                            FROM    e
                            WHERE   (e.in_i, e.in_j) = (g.in_i, g.in_j))
       GROUP BY (g.in_i, g.in_j), g.fanout
       HAVING COUNT(*) = g.fanout
      ) AS go(in_i, in_j, ret)
)
```

Figure 10: CTE returns with usage of the call graph hash table. Used in dtw default with call graph hash table

dtw reference

```
CREATE FUNCTION f(args)
RETURNS DOUBLE PRECISION
WITH RECURSIVE
    call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
    (See figure 3),
    base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS
    (See figure 12),
    evaluation(in_i, in_j, val, ref, ref_site, ref_fanout) AS
    (See figure 13)
SELECT  e.val
FROM    evaluation AS e
WHERE   (e.in_i, e.in_j) = ((dtw.args).i, (dtw.args).j);
$$ LANGUAGE SQL STABLE
                STRICT;
d
```

Figure 11: Function body of dtw reference

```
base_cases(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (
    SELECT g.in_i, g.in_j, g.val, g_ref.in_i, g_ref.in_j, g_ref.site,
    g_ref.fanout
    FROM call_graph AS g,
         call_graph AS g_ref
    WHERE g.fanout = 0
      AND (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
      AND (g.in_i, g.in_j) = (g_ref.out_i, g_ref.out_j)
),
```

Figure 12: CTE `base_cases`, used in all dtw reference and memoization versions, that use no call graph hash table

```
evaluation(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (
    TABLE base_cases
    UNION ALL
    (
    WITH e AS (TABLE evaluation),
         returns(in_i, in_j, val) AS (
             SELECT go.in_i,
                    go.in_j,
                    (
                    SELECT
                        CASE
                        WHEN go.in_i = 0 AND go.in_j = 0
                        THEN 0 :: double precision
                        WHEN go.in_i = 0 OR go.in_j = 0
                        THEN 'infinity' :: double precision
                        ELSE (SELECT abs(Z.x - Z.y)
                            + LEAST(( SELECT e.val
                                      FROM e
```

```
19                                          WHERE (e.in_i, e.in_j)
20                                              = (go.in_i - 1, go.in_j - 1)
21                                            AND (e.ref_i, e.ref_j)
22                                              = (go.in_i, go.in_j)
23                                            AND e.ref_site = 1),
24                                        ( SELECT e.val
25                                          FROM e
26                                          WHERE (e.in_i, e.in_j)
27                                              = (go.in_i - 1, go.in_j)
28                                            AND (e.ref_i, e.ref_j)
29                                              = (go.in_i, go.in_j)
30                                            AND e.ref_site = 2),
31                                        ( SELECT e.val
32                                          FROM e
33                                          WHERE (e.in_i, e.in_j)
34                                              = (go.in_i, go.in_j - 1)
35                                            AND (e.ref_i, e.ref_j)
36                                              = (go.in_i, go.in_j)
37                                            AND e.ref_site = 3))
38                                   FROM (X JOIN Y ON
39                                        ((X.t, Y.t) = (go.in_i, go.in_j))
40                                        ) AS Z)
41                                    END)
42                  FROM (
43                            SELECT e.ref_i, e.ref_j
44                            FROM e
45                            GROUP BY (e.ref_i, e.ref_j), e.ref_fanout
46                            HAVING COUNT(*) = e.ref_fanout
47                       ) AS go(in_i, in_j)
48                  )
49              SELECT *
50              FROM e
51              WHERE (e.in_i, e.in_j) <> ((dtw.args).i, (dtw.args).j)
52                AND NOT EXISTS(SELECT
53                               FROM returns AS r
54                               WHERE (r.in_i, r.in_j) = (e.ref_i, e.ref_j))
55              UNION ALL
56              SELECT r.in_i, r.in_j, r.val, g.in_i, g.in_j, g.site, g.fanout
57              FROM returns AS r,
58                   call_graph AS g
59              WHERE (r.in_i, r.in_j) = (g.out_i, g.out_j)
60          )
61      )
```

Figure 13: CTE `evaluation`, used in dtw reference and dtw memoization with no hash tables

dtw reference with call graph hash table

```
1  CREATE FUNCTION f(args)
2  RETURNS DOUBLE PRECISION
3  AS $$
4   SELECT prepareHT(CG_HT_ID, 2,    NULL::INT,
5                                    NULL::INT,
6                                    NULL::CALLGRAPHVAL[]);
7  WITH RECURSIVE
8      call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
9      (See figure 3),
10     fill_hashtable AS
11     (See figure 8),
12     base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS
13     (See figure 15),
14     evaluation(in_i, in_j, val, ref, ref_site, ref_fanout) AS
15     (See figure 16)
16 SELECT  e.val
17 FROM    evaluation AS e
18 WHERE   (e.in_i, e.in_j) = ((dtw.args).i, (dtw.args).j);
19 $$ LANGUAGE SQL STABLE
20                 STRICT;
```

Figure 14: Function body of dtw reference with call graph hash table. Red areas mark the modifications compared to the original version

```
1  base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS (
2      SELECT  g.in_i, g.in_j, g.val,
3              g_ref.in_i, g_ref.in_j, g_ref.site, g_ref.fanout
4      FROM    call_graph AS g,
5              (SELECT COUNT(*) FROM fill_hashtable) AS _,
6              lookupHTRecord(CG_HT_ID, false, g.in_i, g.in_j)
7                  AS ht(out_i INT, out_j INT, cgval CALLGRAPHVAL[]),
8              unnest(ht.cgval) AS g_ref
9      WHERE   g.fanout = 0
10       AND   (g_ref.fanout > 0 OR g_ref.fanout IS NULL)
11 ),
```

Figure 15: CTE `base_cases`, used in all dtw reference and memoization versions, that use the call graph hash table

```
1  evaluation(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (
2      TABLE base_cases
3      UNION ALL
4      (
5          WITH e AS (TABLE evaluation),
6               returns(in_i, in_j, val) AS (...)
7          SELECT  *
8          FROM    e
9          WHERE   (e.in_i, e.in_j) <> ((dtw.args).i, (dtw.args).j)
10                 AND NOT EXISTS( SELECT
```

```
11                                  FROM    returns AS r
12                                  WHERE   (r.in_i, r.in_j)
13                                        = (e.ref_i, e.ref_j))
14          UNION ALL
15          SELECT  r.in_i, r.in_j, r.val, g.in_i, g.in_j, g.site, g.fanout
16          FROM    returns AS r,
17                  lookupHTRecord(CG_HT_ID, false, r.in_i, r.in_j)
18                     AS ht(out_i INT, out_j INT, cgval CALLGRAPHVAL[]),
19                  unnest(ht.cgval) AS g
20      ))
```

Figure 16: CTE `evaluation`, used in dtw reference with hash table

dtw reference with result hash table

```
1   CREATE FUNCTION f(args)
2   RETURNS DOUBLE PRECISION
3   AS $$
4   SELECT prepareHT(R_HT_ID,  2,    NULL::INT,
5                                    NULL::INT,
6                                    NULL::DOUBLE PRECISION);
7   WITH RECURSIVE
8       call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
9       (See figure 18),
10      base_cases(in_i, in_j, ref, ref_site, ref_fanout) AS
11      (See figure 12),
12      evaluation(in_i, in_j, ref, ref_site, ref_fanout) AS
13      (See figure 19)
14  SELECT  result_ht.val
15  FROM    (SELECT COUNT(*) FROM evaluation) AS _,
16          lookupHTRecord(R_HT_ID, false, (dtw.args).i, (dtw.args).j)
17             AS result_ht(i INT, j INT, val DOUBLE PRECISION);
18  $$ LANGUAGE SQL STABLE
19                  STRICT;
```

Figure 17: Function body of dtw reference with call graph hash table. Red areas mark the modifications compared to the original version

```
1   call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS (
2       SELECT (dtw.args).i, (dtw.args).j, NULL :: INT, NULL :: BIGINT,
3              (dtw.args).i, (dtw.args).j, NULL :: DOUBLE PRECISION
4       UNION
5       SELECT  g out_i, g.out_j, edges.*
6       FROM    call_graph AS g,
7               LATERAL (
8                WITH slices(site, out) AS (...),
9                     calls(site, fanout, i, j, val) AS (...)
10                TABLE calls
11               UNION ALL
12               SELECT NULL :: INT, 0, g out_i, g.out_j,
13                      CASE
```

```
14                      WHEN insertToHT(R_HT_ID, true, g out_i, g.out_j,(
15                          SELECT  CASE
16                                  WHEN g.out_i = 0 AND g.out_j = 0
17                                  THEN 0.0
18                                  WHEN g.out_i = 0 OR g.out_j = 0
19                                  THEN 'infinity' :: DOUBLE PRECISION
20                                  ELSE (SELECT abs(Z.x - Z.y) +
21                                               LEAST(NULL, NULL, NULL)
22                                        FROM   (X JOIN Y ON
23                                               ((X.t, Y.t)
24                                               = (g out_i, out_j)))
25                                                   AS Z)
26                                  END)) IS NOT NULL
27                      THEN NULL
28                      END
29              WHERE NOT EXISTS(TABLE calls)
30              ) AS edges(site, fanout, i, j, val)
31  ),
```

Figure 18: CTE `call_graph`, used in all dtw reference versions with result hash table

```
1   evaluation(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (
2       TABLE base_cases
3       UNION ALL
4       (
5           WITH e AS (TABLE evaluation),
6                returns(in_i, in_j, empty) AS (See figure 20)
7           SELECT  *
8           FROM    (SELECT COUNT(*) FROM returns) AS _,
9                   e
10          WHERE   (e.in_i, e.in_j) <> ((dtw.args).i, (dtw.args).j)
11                  AND NOT EXISTS(SELECT
12                                 FROM  lookUpHT(R_HT_ID, false,
13                                                e.ref_i, e.ref_j)
14                                       AS ht(i INT, j INT,
15                                             val DOUBLE PRECISION)))
16          UNION ALL
17          SELECT  r.in_i, r_in_j, g.in_i, g.in_j, g.site, g.fanout
18          FROM    returns AS r,
19                  call_graph AS g
20          WHERE   (r.in_i, r.in_j) = (g.out_i, g.out_j)
21      )
22  )
```

Figure 19: CTE `evaluation`, used in all dtw reference versions with result hash table

```
returns(in_i, in_j, ht) AS (
 SELECT go.in_i,
        go.in_j,
        insertToHT(R_HT_ID, true, go.in_i, go.in_j,
          (SELECT CASE
                  WHEN go.in_i = 0 AND go.in_j = 0
                  THEN 0.0
                  WHEN go.in_i = 0 OR go.in_j = 0
                  THEN 'infinity' :: DOUBLE PRECISION
                  ELSE (SELECT abs(Z.x - Z.y)
                               +LEAST((SELECT result_ht.val
                                       FROM   lookUpHTRecord(
                                              R_HT_ID,
                                              false,
                                              go.in_i - 1,
                                              go.in_j - 1)
                                          AS result_ht(
                                           i INT, j INT,
                                           val DOUBLE PRECISION)),
                                      (SELECT result_ht.val
                                       FROM   lookUpHTRecord(
                                              R_HT_ID,
                                              false,
                                              go.in_i - 1,
                                              go.in_j)
                                          AS result_ht(
                                           i INT, j INT,
                                           val DOUBLE PRECISION)),
                                      (SELECT result_ht.val
                                       FROM   lookUpHTRecord(
                                              R_HT_ID,
                                              false,
                                              go.in_i,
                                              go.in_j - 1)
                                          AS result_ht(
                                           i INT, j INT,
                                           val DOUBLE PRECISION)))
                        FROM (X JOIN Y ON ((X.t, Y.t) = (go.in_i, go.in_j)
                             )) AS Z)
                  END))
 FROM (
          SELECT    e.ref
          FROM      e
          GROUP BY (e.ref), e.ref_fanout
          HAVING COUNT(*) = e.ref_fanout
      ) AS go(in_i, in_j)
)
```

Figure 20: CTE `returns`, used in all dtw reference versions with result hash table

dtw reference with call graph and result hash tables

```
1  CREATE FUNCTION f(args)
2  RETURNS DOUBLE PRECISION
3  AS $$
4  SELECT prepareHT(CG_HT_ID, 2,    NULL::INT,
5                                   NULL::INT,
6                                   NULL::CALLGRAPHVAL[]),
7         prepareHT(R_HT_ID,  2,    NULL::INT,
8                                   NULL::INT,
9                                   NULL::DOUBLE PRECISION);
10 WITH RECURSIVE
11     call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
12     (See figure 18),
13     fill_hashtable AS
14     (See figure 8),
15     base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS
16     (See figure 15),
17     evaluation(in_i, in_j, ref, ref_site, ref_fanout) AS
18     (See figure 22)
19 SELECT  result_ht.val
20 FROM    (SELECT COUNT(*) FROM evaluation) AS _,
21         lookupHTRecord(R_HT_ID, false, (dtw.args).i, (dtw.args).j)
22            AS result_ht(i INT, j INT, val DOUBLE PRECISION);
23 $$ LANGUAGE SQL STABLE
24                 STRICT;
```

Figure 21: Function body of dtw reference with call graph and result hash tables

```
1  evaluation(in_i, in_j, val, ref_i, ref_j, ref_site, ref_fanout) AS (
2      TABLE base_cases
3      UNION ALL
4      (    WITH e AS (TABLE evaluation),
5               returns(in_i, in_j, ht) AS (See figure 20)
6          SELECT  *
7          FROM    (SELECT COUNT(*) FROM returns) AS _,
8                  e
9          WHERE   (e.in_i, e.in_j) <> ((dtw.args).i, (dtw.args).j)
10                 AND NOT EXISTS(SELECT
11                                FROM  lookUpHT(R_HT_ID, false,
12                                               e.ref_i, e.ref_j)
13                                   AS ht(i INT, j INT,
14                                         val DOUBLE PRECISION)))
15         UNION ALL
16         SELECT  r.in_i, r_in_j, g.in_i, g.in_j, g.site, g.fanout
17         FROM    returns AS r,
18                 lookupHTRecord(CG_HT_ID, false, r.in_i, r.in_j)
19                    AS ht(out_i INT, out_j INT, cgval CALLGRAPHVAL[]),
20                 unnest(ht.cgval) AS g ))
```

Figure 22: CTE `evaluation` used in all dtw reference and memoization versions with call graph and result hash tables

dtw memoization

```
CREATE TABLE IF NOT EXISTS memoization_dtw
(
    in_i INT,
    in_j INT,
    val  double precision,
    PRIMARY KEY (in_i, in_j)
);
```

Figure 23: Table creation of dtw memoization table

```
CREATE FUNCTION f(args)
RETURNS DOUBLE PRECISION
WITH RECURSIVE
    call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
    (See figure 25),
    base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS
    (See figure 12),
    evaluation(in_i, in_j, val, ref, ref_site, ref_fanout) AS
    (See figure 13),
    memoization AS (
        INSERT INTO memoization_dtw
            SELECT DISTINCT ON (e.in_i, e.in_j) e.in_i, e.in_j, e.val
            FROM evaluation AS e
            ON CONFLICT DO NOTHING
    )
SELECT  e.val
FROM    evaluation AS e
WHERE   (e.in_i, e.in_j) = ((dtw.args).i, (dtw.args).j);
$$ LANGUAGE SQL VOLATILE
                STRICT;
```

Figure 24: Function body of dtw memoization

```
call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS (
    SELECT (dtw.args).i,
           (dtw.args).j,
           NULL :: INT,
           NULL :: BIGINT,
           (dtw.args).i,
           (dtw.args).j,
           NULL :: DOUBLE PRECISION

    UNION

    SELECT g.out_i, out_j, edges.*
    FROM call_graph AS g,
     LATERAL (
         WITH memoization(site, fanout, i, j, val) AS (
             SELECT NULL :: INT, 0, m.in_i, m.in_j, m.val
```

```
17              FROM memoization_dtw AS m
18              WHERE (g.out_i, g.out_j) = (m.in_i, m.in_j)
19          ),
20               slices(site, out_i, out_j) AS (...),
21               calls(site, fanout, i, j, val) AS (
22                   SELECT    s.site,
23                             COUNT(*) OVER (),
24                             (s.out).args.i,
25                             (s.out).args.j,
26                             NULL :: DOUBLE PRECISION
27                   FROM      slices AS s
28                   WHERE     (s.out).not_bottom
29               )
30          TABLE memoization
31
32          UNION ALL
33
34          SELECT *
35          FROM   calls
36          WHERE  NOT EXISTS(TABLE memoization)
37
38          UNION ALL
39
40          SELECT NULL :: INT,
41                 0,
42                 g.out_i, g.out_j,
43                 (
44                     SELECT CASE
45                                WHEN g.out_i = 0 AND g.out_j = 0
46                                THEN 0 :: DOUBLE PRECISION
47                                WHEN g.out_i = 0 OR g.out_j = 0
48                                THEN 'infinity' :: DOUBLE PRECISION
49                                ELSE (SELECT abs(Z.x - Z.y) +
50                                             LEAST(NULL::DOUBLE PRECISION,
51                                                   NULL::DOUBLE PRECISION,
52                                                   NULL::DOUBLE PRECISION)
53                                      FROM (X JOIN Y ON
54                                           ((X.t, Y.t)
55                                           = (g.out_i, g.out_j))
56                                           ) AS Z)
57                                END)
58          WHERE NOT EXISTS(TABLE memoization)
59            AND NOT EXISTS(TABLE calls)
60          ) AS edges(site, fanout, i, j, val)
61  ),
```

Figure 25: CTE `call_graph`, used in dtw memoization

dtw memoization with call graph and result hash table and table memoization

```
CREATE FUNCTION f(args)
RETURNS DOUBLE PRECISION
AS $$
SELECT prepareHT(CG_HT_ID, 2,    NULL::INT,
                                 NULL::INT,
                                 NULL::CALLGRAPHVAL[]),
       prepareHT(R_HT_ID,  2,    NULL::INT,
                                 NULL::INT,
                                 NULL::DOUBLE PRECISION);
WITH RECURSIVE
    call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
    (See figure 29),
    fill_hashtable AS
    (See figure 8),
    base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS
    (See figure 15),
    evaluation(in_i, in_j, ref, ref_site, ref_fanout) AS
    (See figure 22),
    memoization AS (
        INSERT INTO memoization_dtw
            SELECT DISTINCT ON (e.in_i, e.in_j) e.in_i, e.in_j, e.val
            FROM evaluation AS e
            ON CONFLICT DO NOTHING
    )
SELECT  result_ht.val
FROM    (SELECT COUNT(*) FROM evaluation) AS _,
        lookupHTRecord(R_HT_ID, false, (dtw.args).i, (dtw.args).j)
          AS result_ht(i INT, j INT, val DOUBLE PRECISION);
$$ LANGUAGE SQL VOLATILE
                STRICT;
```

Figure 26: Function body of dtw memoization with call graph and result hash tables. It uses the memoization table for memoization

```
call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS (
    SELECT (dtw.args).i,
           (dtw.args).j,
           NULL :: INT,
           NULL :: BIGINT,
           (dtw.args).i,
           (dtw.args).j,
           NULL :: DOUBLE PRECISION

    UNION

    SELECT g.out_i, out_j, edges.*
    FROM call_graph AS g,
     LATERAL (
```

```
15          WITH memoization(site, fanout, i, j, val) AS (
16              SELECT NULL :: INT, 0, m.in_i, m.in_j, m.val
17              FROM memoization_dtw AS m
18              WHERE (g.out_i, g.out_j) = (m.in_i, m.in_j)
19          ),
20               slices(site, out_i, out_j) AS (...),
21               calls(site, fanout, i, j, val) AS (
22                   SELECT    s.site,
23                             COUNT(*) OVER (),
24                             (s.out).args.i,
25                             (s.out).args.j,
26                             NULL :: DOUBLE PRECISION
27                   FROM      slices AS s
28                   WHERE     (s.out).not_bottom
29               )
30          TABLE memoization
31
32          UNION ALL
33
34          SELECT *
35          FROM   calls
36          WHERE  NOT EXISTS(TABLE memoization)
37
38          UNION ALL
39
40          SELECT NULL :: INT, 0, g out_i, g.out_j,
41                 CASE
42                 WHEN insertToHT(R_HT_ID, true, g out_i, g.out_j,(
43                     SELECT  CASE
44                             WHEN g.out_i = 0 AND g.out_j = 0
45                             THEN 0.0
46                             WHEN g.out_i = 0 OR g.out_j = 0
47                             THEN 'infinity' :: DOUBLE PRECISION
48                             ELSE (SELECT abs(Z.x - Z.y) +
49                                          LEAST(NULL, NULL, NULL)
50                                   FROM   (X JOIN Y ON
51                                          ((X.t, Y.t)
52                                          = (g out_i, out_j)))
53                                              AS Z)
54                             END)) IS NOT NULL
55                 THEN NULL
56                 END
57          WHERE NOT EXISTS(TABLE calls)
58          ) AS edges(site, fanout, i, j, val)
59  ),
```

Figure 27: CTE `call_graph`, used in dtw memoization with hash tables and table memoization

dtw memoization with call graph and result hash table and hash table memoization

```
CREATE FUNCTION f(args)
RETURNS DOUBLE PRECISION
AS $$
SELECT prepareHT(CG_HT_ID, 2,    NULL::INT,
                                 NULL::INT,
                                 NULL::CALLGRAPHVAL[]),
       prepareHT(R_HT_ID,  2,    NULL::INT,
                                 NULL::INT,
                                 NULL::DOUBLE PRECISION);
WITH RECURSIVE
    call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS
    (See figure 29),
    fill_hashtable AS
    (See figure 8),
    base_cases(in_i, in_j, val, ref, ref_site, ref_fanout) AS
    (See figure 15),
    evaluation(in_i, in_j, ref, ref_site, ref_fanout) AS
    (See figure 22),
SELECT  result_ht.val
FROM    (SELECT COUNT(*) FROM evaluation) AS _,
        lookupHTRecord(R_HT_ID, false, (dtw.args).i, (dtw.args).j)
          AS result_ht(i INT, j INT, val DOUBLE PRECISION);
$$ LANGUAGE SQL STABLE
                STRICT;
```

Figure 28: Function body of dtw memoization with call graph and result hash tables. It uses the result hash table for memoization

```
call_graph(in_i, in_j, site, fanout, out_i, out_j, val) AS (
    SELECT (dtw.args).i,
           (dtw.args).j,
           NULL :: INT,
           NULL :: BIGINT,
           (dtw.args).i,
           (dtw.args).j,
           NULL :: DOUBLE PRECISION

    UNION

    SELECT g.out_i, out_j, edges.*
    FROM call_graph AS g,
     LATERAL (
         WITH memoization(site, fanout, i, j, val) AS (
             SELECT NULL :: INT, 0, ht.i, ht.j, null :: DOUBLE PRECISION
             FROM   lookupHT(R_HT_ID, false, g.out_i, g.out_j)
                        AS ht(i INT, j INT, val DOUBLE PRECISION)
         ),
              slices(site, out_i, out_j) AS (...),
```

```
21                calls(site, fanout, i, j, val) AS (
22                    SELECT    s.site,
23                              COUNT(*) OVER (),
24                              (s.out).args.i,
25                              (s.out).args.j,
26                              NULL :: DOUBLE PRECISION
27                    FROM      slices AS s
28                    WHERE     (s.out).not_bottom
29                )
30            TABLE memoization
31
32            UNION ALL
33
34            SELECT *
35            FROM   calls
36            WHERE  NOT EXISTS(TABLE memoization)
37
38            UNION ALL
39
40            SELECT NULL :: INT, 0, g out_i, g.out_j,
41                   CASE
42                   WHEN insertToHT(R_HT_ID, true, g out_i, g.out_j,(
43                       SELECT  CASE
44                               WHEN g.out_i = 0 AND g.out_j = 0
45                               THEN 0.0
46                               WHEN g.out_i = 0 OR g.out_j = 0
47                               THEN 'infinity' :: DOUBLE PRECISION
48                               ELSE (SELECT abs(Z.x - Z.y) +
49                                            LEAST(NULL, NULL, NULL)
50                                     FROM   (X JOIN Y ON
51                                            ((X.t, Y.t)
52                                            = (g out_i, out_j)))
53                                                AS Z)
54                               END)) IS NOT NULL
55                   THEN NULL
56                   END
57            WHERE NOT EXISTS(TABLE calls)
58            ) AS edges(site, fanout, i, j, val)
59    ),
```

Figure 29: CTE `call_graph`, used in all dtw memoization versions with both hash tables and result hash table memoization